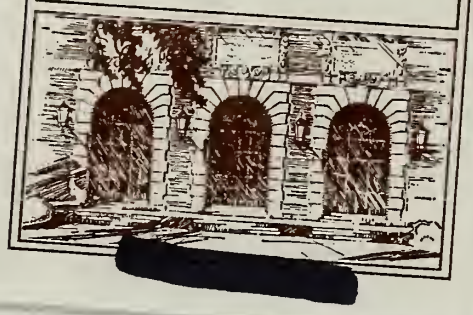
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
I26r
no.685-690
cop.2



I26n UIUCDCS-R-75-685
no.685

AN INTERACTIVE COBOL SYSTEM FOR PLATO

by

Robert D. Barnett

The Library of the
JAN 1 1975
University of
at Urbana-Champaign

January, 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

15

UIUCDCS-R-75-685

AN INTERACTIVE COBOL SYSTEM FOR PLATO

by

Robert D. Barnett

January, 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science,
January, 1975

Digitized by the Internet Archive
in 2013

http://archive.org/details/interactivecobol685barn

51084
I16r
no 685-688
cop.2

ACKNOWLEDGMENT

The author would like to express appreciation to Dr. Thomas R. Wilcox for introducing the author to the ACSES project and for his guidance in the preparation of this thesis, and to Dr. Jurg Nievergelt for his advice and encouragement. Appreciation is also expressed to the other members of the compiler group, Mike Tindall, Al Davis and Mike Milner, for their advice and cooperation.

The ACSES project in PLATO is supported by grant NSF-EC-41511 of the National Science Foundation. The author has been supported in part by Veterans' Bill benefits.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 The PLATO system	1
	1.2 General structure	2
2	THE COBOL SUBSET	4
3	THE PROGRAM EDITOR	7
	3.1 As seen by the student	7
	3.2 The tables of the compiler-writing system	7
	3.3 Internal character codes	8
	3.4 Columnar format and tab settings	9
	3.5 Lexical tables	10
	3.6 Key function tables	17
	3.7 Predefined symbols	17
	3.8 Syntax description	19
4	THE EXECUTION INTERPRETER	21
	4.1 Primary data structures	22
	4.2 The supervisor	24
	4.2.1 Interaction with the student.	24
	4.2.2 Expression evaluation	26
	4.2.3 The PERFORM statement	27
	4.2.4 The GO TO statement	28
	4.2.5 Other statements	29
	4.2.6 Unit "address"	29
	4.3 The machine	31
	4.3.1 Binary mode	34
	4.3.2 Decimal mode	36
	4.3.3 Character mode	36
5	DOCUMENTATION PROVISIONS	37
6	CONCLUSIONS	40
LIST OF REFERENCES		42

Page

APPENDIX

A COBOL HELP LESSON. 44

B COBOL TABLES IN THE COMPILER-WRITING SYSTEM . 65

B.1 Primary tables 66

B.1.1 Internal character codes . . . 67
B.1.2 Initial tab settings. 68
B.1.3 Key function tables 69
B.1.4 Lexical character classes . . . 77
B.1.5 Lexical node table 78
B.1.6 Safe nodes 83
B.1.7 Predefined symbols 84
B.1.8 Field tokens 91
B.1.9 New symbol allocation classes. . 92

B.2 Picture tables 93

B.2.1 Lexical character classes . . . 94
B.2.2 Lexical node table 95
B.2.3 Safe nodes 99

C OTHER DOCUMENTATION 100

C.1 Transition diagram sample 101
C.2 Flowchart of "cobolrun" 104
C.3 Mode table diagram. 128

D SAMPLE COBOL PROGRAM. 129

E PROGRAM LISTINGS 136

1. INTRODUCTION

A project to provide automated instruction in computer science, ACSES (Automatic Computer Science Education System)[8], is in progress at the University of Illinois at Urbana-Champaign using the PLATO computer-based education system. An important part of ACSES is a set of interactive compilers which allow students learning one of several programming languages to write and execute small programs through their PLATO terminal. This thesis describes the design and implementation of the COBOL compiler system in ACSES.

1.1 The PLATO system

PLATO is a computer-based education system which has evolved since 1959 [7]. The current system, PLATO IV, includes a CDC CYBER 73 computer linked with over 700 terminals in more than 100 locations. A hierarchy of memories (fast core memory, extended core storage and disks) contain lesson material for a wide variety of academic areas. The PLATO IV student terminal has a keyboard input device that includes regular typewriter keys and special function keys. The terminal screen is an 8 1/2" square plasma display panel, providing storage of information on the screen itself without flicker and without being constantly refreshed [9]. A powerful language, TUTOR, is used to write the instructional programs, called "lessons."

1.2 General structure

The COBOL compiler system consists of three TUTOR lessons. The COBOL editor accepts the program entered by the student using the PLATO keyboard. The COBOL interpreter executes the student's COBOL program. A third lesson provides information needed to successfully use the COBOL system, such as the subset of COBOL implemented.

Since it is a part of ACSES, the COBOL system is designed for use by students beginning their study of COBOL. Therefore a primary goal has been to provide the student with good diagnostic messages, issued as soon as possible, both in the editor and the interpreter. The student can write a short COBOL program and execute it within a few minutes.

Only a relatively small subset of COBOL has been implemented. This subset, described in chapter two, is designed to be adequate for beginning COBOL programmers.

A key feature of the COBOL system is that it uses a table-driven compiler-writing system developed by Dr. Thomas Wilcox and Michael Tindall [13]. PL/I has already been implemented using this system, while FORTRAN IV and BASIC are currently being developed. The use of this language-independent system greatly reduces the effort required to develop an editor for a specific language, since the language implementor only needs to add the specifics of a language to a skeleton editor.

This consists primarily of specifying table entries and writing a description of the language's syntax. However, the COBOL interpreter, while using some parts of the compiler-writing system, is largely language-dependent.

2. THE COBOL SUBSET

Full COBOL is an extensive programming language designed for data processing applications. To implement the entire COBOL language would greatly exceed the table sizes of the language-independent editor routines. However a relatively small subset of COBOL, given in Appendix A, is sufficient to satisfy the educational goals of ACSES with respect to COBOL. The overall criterion for selecting the subset has been to include all the features essential to writing elementary COBOL programs.

The American National Standards Institute (ANSI), formerly known as the United States of America Standards Institute (USASI), is responsible for maintaining standards for COBOL. It has defined a minimum subset of COBOL which manufacturers are expected to include in their COBOL compilers [10]. This minimum ANSI subset is comparable to the subset described here. However several features have been deleted and a few added, for reasons given in the following paragraphs.

Any COBOL compiler must support the basic data processing requirements. These include reading and writing data organized in records and files, moving and comparing data, doing simple arithmetic calculations, and controlling the flow of program execution. The COBOL statements OPEN, READ, WRITE,

CLOSE, MOVE, IF, ADD, SUBTRACT, MULTIPLY, DIVIDE, GO, PERFORM, EXIT and STOP form the minimum set that satisfies these requirements. All are included in the ANSI subset and all except DIVIDE are in the PLATO subset.

Compactness is a desirable quality of computer programs [11] which COBOL programs seldom achieve due to the requirements of COBOL syntax. In order to achieve some measure of compactness, the COMPUTE verb, not included in the ANSI subset, has been included in the PLATO subset. Without the COMPUTE verb a COBOL programmer must use several arithmetic statements and must define intermediate results in order to evaluate an arithmetic expression. The DIVIDE verb has been excluded, since it is infrequently used and COMPUTE includes its functions.

Other extensions to the ANSI subset which improve the compactness and locality of COBOL are nested IF statements, some forms of compound conditions, and arithmetic expressions in conditions.

Another influence in the definition of the COBOL subset was Armstrong [1], who points out the problems with the ALTER verb and with complex conditions. ALTER often hides program structure by causing a GO to branch to a paragraph other than that named. Instead the DEPENDING ON option of the GO TO has been included. Beginning COBOL programmers often are confused by the rather complex rules governing precedence of

logical operators and governing implied operators and subjects of relations. To prevent these problems, implied operators and subjects are not allowed, the logical connectives AND and OR may not be used in the same condition, and NOT is restricted to use as a part of a relation or class test.

The language-independent editor routines are designed for programs not larger than 28 lines of 64 characters each (the space available on the PLATO screen). Since COBOL requires many lines of code for even a trivial program, modifications to COBOL were required to reduce program sizes. The first two of COBOL's four divisions, the IDENTIFICATION and ENVIRONMENT divisions, have been excluded, as well as the two headers usually beginning the DATA division. This is not a serious loss, since most of the excluded portions are fixed division and section headers which the student can easily learn elsewhere.

The final criterion for inclusion in the PLATO subset was space availability. Each table in the editor has a maximum size; the subset implemented uses nearly the maximum number of predefined names and entries in the syntax code table. For this reason some lesser used features that are part of the ANSI standard subset could not be included, such as the EXAMINE verb and index data-items.

3. THE PROGRAM EDITOR

3.1 As seen by the student

As the student types his program using the PLATO keyboard, it is displayed on the PLATO screen. At the same time, it is examined by the lexical, syntactic and semantic routines of the editor. Any errors detected are immediately called to the student's attention by drawing a rectangle around the character or word just entered. If the student then presses the HELP key, an error message is displayed explaining the error or listing legal alternatives. If the student again presses HELP and the error is syntactic or semantic, a message giving the syntactic type of the last token entered is written.

By using the special black keys of the PLATO keyboard, the student can position the cursor at any point in his program, add or delete characters, and indicate when he has completed his program.

3.2 The tables of the compiler-writing system

The editing features described in Section 3.1 are features of the compiler-writing system incorporated into the COBOL editor. A set of tables, developed and maintained by

auxiliary programs, describes the internal character codes, key function tables, default keyboard tab stops, lexical class assignments, lexical node table, "safe" nodes, predefined symbols, field symbols, new symbol allocation classes and syntax tables. Details of each of these tables as used in the COBOL editor are given in the following sections.

In addition to the tables, semantic routines and error messages are added to the editor by the language implementor.

Although initially developed for free format languages, the compiler-writing system has been modified to accomodate features of fixed format languages, such as the continuation of words between lines in COBOL. In general the system works well with COBOL, but there are problems with lexical analysis, discussed in Section 3.5. Semantic routines, written in TUTOR, must be used extensively, particularly in parsing data structures and in examining pictures, to do verifications which are difficult or impossible through the tables.

3.3 Internal character codes

There are 63 characters available, as given in Appendix B.1.1. These are the normally printable characters from the EBCDIC character codes used in the IBM 360/370 and

other computers. This set was chosen to avoid confusion for the students using PLATO who are also using the IBM computers available on campus and elsewhere. Although the bit configurations are different, the characters have the same ordering as in EBCDIC.

In the character tables seven bits are available for each character. The extra seventh bit is used in the interpreter as a sign bit of a signed decimal field. If the high-order bit of the most significant digit position is on, the field is negative.

There is a sixty-fourth character, made of three question marks, which the student cannot enter. At the beginning of execution the interpreter sets all character positions to this value. Then the arithmetic routines check for it to determine whether a field has ever been assigned a value.

3.4 Columnar format and tab settings

Standard COBOL has a fixed column format based on an 80 column card, but the PLATO screen is only 64 characters wide. To make the most efficient use of the available space, those fields used for documentation only, usually columns one to six and 73 to 80, are not available. In the resulting format column one may contain a hyphen to indicate continuation, an asterisk to indicate a line of comments, or a blank. Columns two through five are area A, where division and

section headers, paragraph names and some level numbers must begin. Area B is columns six through 64.

During editing tab stops appear near the top of the screen. The student may press TAB to have blanks inserted up to the next tab stop. The initial settings of the tab stops are in a table, but may be changed by the student.

The first tab stop setting in COBOL is at column six, so that by pressing the TAB key at the start of a new line the student can quickly get to area B. Additional tab stops provide for easy identification or organization of clauses in columns, as given in Appendix B.1.2.

3.5 Lexical tables

The lexical tables include the lexical class assignment table, the lexical node table and the safe node list. Using these tables the language-independent editor performs the lexical analysis of the COBOL program, consisting of scanning the input and constructing the tokens of the language. COBOL tokens may be words, punctuation, arithmetic operators, relational operators and picture strings.

The model for lexical analysis is a finite state machine. Lexical analysis of a particular token starts in state zero, alias node zero, then node transitions are made according to current node and character class of the next input, as established by the lexical class assignment table. The column on the screen in which the character appears is also assigned a character class so that lexical decisions may be

based on the screen location of a character. Table entries representing illegal inputs at a given node contain error message numbers. When a legal token is recognized, the table entry indicates whether the token is extendable (an extendable token is one whose termination is determined only by a delimeter not included in the token). There are also provisions to generate field tokens indicating the presence of a token in a particular range of columns, to allow for tokens to be continued between lines and to call a subroutine of nodes.

COBOL presented several problems in adopting it to use the lexical tables. Largest of these was that the lexical characteristics of picture strings are vastly different from that of all other COBOL tokens, requiring a far greater number of node table entries than are available in a single node table. Minor modifications were made to the editor which allow for a second set of lexical tables for picture strings which are loaded in place of the regular set when needed. Both sets of lexical tables are discussed in the following paragraphs.

The first set of lexical tables is used for all tokens except picture strings. The lexical class assignments, given in Appendix B.1.4, partition the characters and columns into classes; all members of a class are equivalent in lexical analysis. The columns are partitioned

into three character classes: column one, two through five and six through 64 representing the continuation column, area A and area B, respectively. The characters P, I and C each belong to a character class consisting only of that character, so that PIC followed by a blank may be recognized lexically and used as described later.

Safe nodes are those which can handle the start of a new line. If the lexical routines are not in a safe node when the student presses CR (carriage return), blanks are inserted on the current line until a safe node is reached. In the COBOL tables the safe nodes, listed in Appendix B.1.6, include the initial node, the node for a string of blanks, the node for a comment line and all nodes representing tokens which may be continued on the next line. An example of the latter is node one; it represents a sequence of characters which forms a COBOL word. The word may be terminated by a delimeter or may be continued on the current line or in area B of the following line.

The first lexical node table uses every feature available to it in the compiler-writing system. Most of the tokens recognized are COBOL words consisting of letters, digits and hyphens. They are always extendable tokens because they are terminated by a delimeter, and they may be continued on a second line. Numeric constants and character string constants may also be continued. A numeric constant is an extendable token. Although a character string constant is terminated

by a quote which is part of the token, it is treated as an extendable token in order to enforce the COBOL rule requiring a delimeter following it.

A problem that became apparent during program testing was partially solved by modifying the lexical tables. It is not difficult to write a COBOL program which fits on the PLATO screen and could be successfully entered by the student if there were only more entries in the intermediate text. The source of the problem is COBOL's verboseness and in particular its rule that blanks separate most tokens, so that about one-half of the intermediate text entries are tokens of one or more blanks. To alleviate this problem the required blank is included as part of the preceding token whenever possible. This can be reasonably accomplished only for tokens which must always be delimited by spaces; this includes the arithmetic operators: +, -, *, /; the relational operators: <, =, >; the parentheses and the period. The lexical node table entries for these tokens define two character nonextendable tokens including the special character and a blank.

A serious problem is caused by the dual role of the period in COBOL as both the decimal point in numeric constants and as the punctuation terminating a sentence. A series of digits followed by a period and a blank is interpreted in COBOL as an integer followed by a punctuation symbol. The problem arises in the lexical node table if the characters scanned so far are one or more digits followed by a period.

It is necessary to inspect the next character to determine the role of the period. If the next character is a space, the period was punctuation; if it's a digit, the period was a decimal point. In the former case the lexical node table entry recognizes an extendable token consisting of an integer and a period. This peculiar token, logically two tokens, is assigned to a unique syntactic class. Some additional code is required throughout both the editor and interpreter for this class.

An unresolved problem caused by the dual role of the period is that two violations of COBOL punctuation rules cannot be detected. The lexical node tables will accept a period illegally surrounded by blanks, and a word followed by a period followed by an integer without any required intervening blanks. Because the lexical node table has no information regarding the previous token, it has no way to tell the role of a period that begins a token and must assume that it could be either punctuation or a decimal point, while only one of these can ever be legal at a given point.

Continuation of words, character strings and numeric constants onto the following line is accomplished using a subroutine within the lexical node table. This subroutine uses the continuation features to form an effective token from the parts on the two lines. It is quite satisfactory, completely complying with COBOL continuation rules.

Two field tokens are generated by the regular node table. One indicates the presence of a word or number be-

ginning in area A (columns two through five) so that the syntax routines can verify that tokens are placed in the proper area. The second field token is generated if a token begins with the characters PIC and blank. This field token triggers the loading of the second set of lexical tables to scan the picture string which must follow. A field token is associated with an entry in the name table but the character string associated with the name table entry is never printed, because the field token exists only as a means of passing information.

In the other set of lexical tables, used to scan picture strings, the finite-state machine model was less than ideal but the compiler-writing system allowed enough flexibility to successfully handle the picture strings.

Picture strings in COBOL may be rather complicated. In order to perform editing of numeric values to be printed a picture string such as -$(3),$$9VB99 might be used. The explicit representation of picture strings included in Appendix E.1 is quite long, requiring the entire node table (32 nodes and 20 character classes) to implement. As a result some legal picture strings, such as those using the symbols A, CR, DB, P and +, are not included in the subset.

This size is a result of the inefficiency of the finite-state model when applied to picture strings. The many legal combinations of picture type (alphanumeric, numeric edited and numeric), floating or replacement character (- $ * Z or none), whether a decimal point has been seen yet, and

whether an S was present can only be represented by a different state (node in the table) for each combination.

When a legal picture string has been recognized, the class of the picture is returned to the syntax routines. Bits are set within the class number to indicate the picture type, whether the picture describes a noninteger numeric item and whether the picture indicates that the item should be completely blank when it has a zero value.

However more information is needed from a picture string. In order to verify that the lengths of different records in the same file and of redefined items are equal the syntax routines must know the lengths specified by picture strings. A semantic routine reexamines the picture strings lexically, determining the lengths as well as other information needed later by the interpreter.

An alternate approach to picture strings is to treat each character in the picture string as a token, putting the burden of the verification of the picture strings on the syntactic routines. However, there are several reasons why this was rejected: It would require a large number of statements in the syntax program causing an overflow of the syntax node table; it would use significantly more entries in the name table, in the symbol table and in the intermediate text; and it would reduce the efficiency of the interpreter because the characters of the picture string would no longer

necessarily be in consecutive character table locations (a fact used by the editing routines).

3.6 Key function tables

The key function table assigns an action to each possible student keypress. Usually the regular typewriter keys cause a character to be added to the program, while the black function keys invoke an editing operation such as erasing a word or backing up a line.

The key function table in the regular set of lexical tables is shown in Appendix B.1.3 along with an explanation of the special function codes. A second key function table is in the set of lexical tables for picture strings and is identical to the first table except for a few of the editing keys. Because the lexical routines of the compiler-writing system back up to the start of a token if the student presses any key that backs up the editor, and the token containing the picture string includes PIC which must be scanned by the regular tables, the editing keys which cause the editor to back up are assigned the function of loading the regular tables. Then the regular tables are used to determine the actual editing function to be performed.

3.7 Predefined symbols

The editor handles the tokens of a language using a name table, symbol table, character table and hash table.

Each different character string recognized as a token by lexical analysis has an entry in the name table. More than one name may be linked to the same symbol table entry, such as COMPUTATIONAL and its abbreviation COMP in COBOL. Also one name table entry may be linked to several symbol table entries. This is used in COBOL for the name FILLER which may be used for different unreferenced parts of data structures.

All COBOL symbols which have meaning without declaration by the programmer are included in the predefined portion of the symbol table. This includes keywords and optional words of the PLATO subset, arithmetic and relational operators, punctuation and field symbols.

A few name table entries were not used by the PLATO subset. In the vacant entries are common COBOL keywords not included in the PLATO subset, so that an appropriate error message can be displayed if the student tries to use one of these words.

The compiler-writing system includes a table to assign each symbol to a procedure block. This feature is designed for block structured languages such as PL/I, but since COBOL has no block structure this table is not used in the COBOL editor.

3.8 Syntax description

The syntax of the COBOL subset is given in a syntax description language designed as part of the compiler-writing system. This language is translated by an auxiliary program into a table of 12-bit entries to be interpreted by the editor as the student enters his program. Features of this language include comparisons based on the token just entered, updating symbol table entries, recursive calls to procedures, specifying error message numbers, and calls to semantic routines.

The COBOL subset syntax language program is included in Appendix E.2 with explanations of the functions of each procedure and many statements. Only a broad view of the structure of the syntax program is presented here.

There is a sharp division in the syntax program corresponding to the two divisions of the COBOL subset, the DATA and PROCEDURE divisions. Except for the initial syntax program (that portion preceeding the first procedure) the syntax of the two divisions is totally separate.

The DATA DIVISION portion consists largely of two procedures "fdproc" and "parsitem". "Fdproc" includes the syntax of the clauses of the FD statement, and calls "parsitem" to parse the record descriptions of the FD. A particular invocation of "parsitem" examines all the items directly subordinate to the same item in the data structure, calling itself recursively for any further sub-items.

COBOL posed several problems in these two procedures. Because the clauses of an FD or a data item description may appear only once and in any order, it would have required excessive space to have separate logic paths for each possibility. Instead at the start of the code for each clause a check of either a symbol table entry or a variable is made to insure that the clause has not already occurred in the current FD or data item description. Because many combinations of data item clauses are invalid there are many checks for illegal combinations. Because computations are not available in the syntax language but are needed to determine the lengths and storage locations of files and data items, several calls to semantic routines written in TUTOR are used. More semantic routines are used to access variables and routines of the editor not otherwise available to the syntax program and to check types in relations and VALUE clauses.

The syntax of the PROCEDURE DIVISION presented fewer problems. The primary procedure, "statelst", contains the basic syntax for each COBOL verb. It is called recursively to parse the list of statements which occur in the true and false branches of IF statements and in the AT END branch of the READ statement. There are procedures to parse conditions, relations and expressions, and to look for a particular operand type such as a numeric data item. Substantial type checking is required in MOVE statements and in relations.

4. THE EXECUTION INTERPRETER

After the student has entered his program he goes to the COBOL interpreter lesson to execute his program. The interpreter acts on an intermediate text representation of the COBOL source program, essentially the program represented as a series of tokens. The interpreter carries out the instructions in the PROCEDURE DIVISION by invoking appropriate TUTOR routines.

The most difficult problem in designing the COBOL interpreter was the lack of commands in TUTOR to directly handle binary coded decimal data. The CDC CYBER 73, and therefore also TUTOR, is organized around a 60 bit word containing a floating-point number, a fixed-point integer or 10 six bit characters. However in order to use characters in computation they must first be converted into fixed-point or floating-point form. COBOL uses binary coded decimal fields extensively in input and output files, and allows the programmer to do arithmetic using these fields.

The solution found to this problem was inspired by Wilcox's division of code generation into two phases, translation and coding [12]. The interpreter is organized in two large parts, which will be called the "supervisor" and the "machine." The supervisor includes the translation of expressions to a postfix representation, the scanning of the

intermediate text, all interaction with the student user, and subroutine calls to the machine. The parameters passed to the machine represent an operation code and addresses in the instruction set of an imaginary computer designed to execute COBOL. This is essentially Wilcox's concept of a "source language machine." The machine then interprets this instruction set.

As the interpreter developed, the division of functions between the parts became less distinct. The supervisor is in control of input-output and branching, using parts of the machine as subroutines. But the division into two parts remains as the fundamental organization of the interpreter.

4.1 Primary data structures

Tying the two parts of the interpreter together is a common data structure, the operand list. Entries in the list are initialized by the supervisor and modified by the machine. An entry in the list contains all information needed by the machine about a particular operand. This may include the data type of the operand, its length in characters, its number of decimal places, its storage location, whether it is a signed field, its binary value, whether its binary value has been determined, the symbol table pointer to the operand and the symbol table pointer of the operand's

picture. The address of a data item within the machine is represented by a pointer to the operand list.

Another important data structure is the symbol table constructed by the COBOL editor. It contains static information about each symbol in the source program, such as its length, location and relationship to other items in a data structure. Details of how the symbol table entries are used for various symbol classes are included in Appendix E.1.; this information is needed because the names of the symbol table entries do not necessarily reflect their usage in COBOL. Anyone examining the lessons in the future will have to refer to documentation to understand many of the statements involving symbol table entries.

In two cases symbol table fields are not static but are used by the supervisor to store information. Fields in the symbol table entry for a file are used to indicate the file status and to hold the offset of the current record from the first character of the block. While executing a PERFORM the symbol table entry of the last paragraph of the PERFORM range is modified to indicate that an exit is needed.

Discussion of other data structures is included with the discussion of the related statement.

4.2 The supervisor

4.2.1 Interaction with the student

To a large degree the COBOL interpreter runs without student intervention. The student is involved in three ways: choosing the execution option, supplying input and responding to error messages.

The student may choose between three methods of execution. One provides a complete trace of the program, with all changes in variables displayed and a box drawn around each paragraph name as it is entered and around each verb as it is executed. A second method includes the statement trace but not the variable trace. The third has no tracing; the only information the student receives about the progress of the program is if a READ, WRITE or STOP is executed or if an error occurs. The full trace method will probably be most often used, since with it a student can see exactly what his COBOL statements are doing.

The supervisor asks the student to enter input data whenever a new block of data is needed by a READ statement. Since blocked input files are included in the subset, not every execution of a READ causes an input request. If the student indicates that there is no more data, the AT END branch of the READ is executed. If he has data to enter, the supervisor asks for input for each elementary item in

each record of the block. To do this the symbol table includes pointers to the "son," "brother" and "father" of each item in the data structure. This structure is taken from Knuth [5] as is the algorithm for traversing the structure to reach every elementary item.

The WRITE statement does not require any student intervention but it is discussed here because it is related to the READ. Because COBOL is often used to generate reports, it is beneficial to be able to see output as it normally appears on a printed output rather than in the few lines available on the same screen display as the COBOL program. Therefore, the number of records per block of an output file is defined as the number of lines per page to be seen on a screen display consisting solely of the student's output. If the student forgot to assign values to any parts of the output record, those parts contain triple question mark characters to alert him to his oversight.

Whenever any part of the COBOL interpreter detects an execution error, an appropriate error message is displayed. The student then has the choice of terminating execution (hopefully to return to the editor to correct his program) or of taking corrective action. For some errors, the corrective action is fixed. For example, the corrective action for a read of an unopened file is to open the file. If the error is an invalid numeric value, the student is asked to

supply a correct value to replace it. This includes correcting an out-of-range subscript or replacing a nonnumeric character in a numeric field. The student may receive help in correcting some errors by pressing the HELP key.

4.2.2 Expression evaluation

In the COBOL subset arithmetic expressions may appear in COMPUTE statements and in conditions in IF and PERFORM statements. The expressions appear in infix order in the intermediate text but must be evaluated according to the precedence rules of the operators. A complication is that the number of decimal places to use for intermediate results can only be determined after the maximum number of decimal places in all operands is known (this problem is discussed further in Section 4.3.1). Therefore, in order to avoid two scans of the intermediate text, one to determine the maximum and another to execute the expression, a postfix representation of the expression is constructed as the expression is scanned from the intermediate text.

The infix to postfix translation uses the operator precedence algorithm given by Gries [2]. The operators are the arithmetic operators +, -, * and /; the relational operators <, = and >; NOT; the NUMERIC test; and assignment. The "f" and "g" precedence values and the machine operation codes corresponding to the operators are stored in otherwise

vacant fields for predefined entries in the symbol table. A stack is used for operators, while operands are added to the previously mentioned operand list.

The postfix representation is a queue of operations to be performed. Each entry in the queue has an operation code and two pointers to the operand list. If one of the operands is an intermediate result, the pointer for it in the operation queue may be wrong. However, the machine can detect the error and correct the pointer.

As each operand is added to the operand list using unit "address" the maximum number of decimal places in any operand is determined. After the entire expression has been scanned the machine is called to execute all of the operations in the queue.

4.2.3 The PERFORM statement

Several options of the PERFORM verb are included in the COBOL subset, requiring a fairly large statement driver in the supervisor. Control of the PERFORM is in the supervisor; the machine is used only to evaluate conditions and determine values.

To save information needed about all PERFORMs currently in effect, a stack is used. Each stack entry includes the first and last paragraph of the PERFORM range, the

paragraph containing the PERFORM, the location of the PERFORM in the intermediate text and, if the TIMES clause was used, the number of iterations remaining.

If a paragraph is the last paragraph in the range of a PERFORM, its symbol table entry contains a pointer to the perform stack. At the end of execution of the paragraph, this pointer should match the current perform stack pointer. If not there must be recursion or overlapping ranges in the PERFORM statements. This error always causes an end to execution, because the supervisor cannot repair an incorrect PERFORM structure. If the pointers match, control is transferred to the PERFORM statement to decide whether another iteration is needed.

This scheme implements the PERFORM verb exactly as in standard COBOL. Nested PERFORMs are valid if their ranges do not overlap and if no paragraph is called recursively.

4.2.4 The GO TO statement

The branching of the COBOL program within the intermediate text is done by the GO TO statement driver in the supervisor. The DEPENDING ON option is implemented using an array, constructed during the interpretation of the GO TO, to save the symbol table pointer of each possible destination which is then indexed by the value of the named variable. This array cannot be replaced by the simple technique of using the index value to locate the intermediate

text entry of the destination, because one or more paragraph names might be continued between lines, causing an extra entry in the intermediate text in addition to the blank token between each paragraph name.

4.2.5 Other statements

The OPEN and CLOSE statements simply set a value in the symbol table entry of the specified file to indicate the file's status. An illegal file status in any of the input-output statements results in an error message. CLOSE also writes any partially full block onto the screen.

The ADD, SUBTRACT, MULTIPLY and MOVE require very little effort in the supervisor since their actual execution is done by the machine. The statement driver routines in the supervisor call the appropriate subroutines needed to organize a machine "instruction," then call the machine with the assembled instruction.

The STOP statement terminates execution of the program. Any symbol table entries that were changed during execution are reset to their initial values. The student can choose to see a page of execution statistics, including such items as how much CPU time and how much storage for data items was used.

4.2.6 Unit "address"

One unit of TUTOR code in the supervisor is so

central to the interpreter that it deserves a detailed discussion. Unit "address" prepares the operand list entries for use by the machine from symbol table entries and subscript values. Its importance is that it organizes information about an operand into an efficient form for use by the machine.

Its most complex function is to determine the offset of each item from the start of the area representing the core of the COBOL machine. Because the character table and the core both contain seven-bit characters, the character table can be used as if it were a part of core. This is used for constants so that, to the machine, constants and variables look almost alike. No core space is wasted duplicating the character table and no extra moves need to be made. The offset and length fields for constants in the operand list are set to exclude the quotes surrounding character strings and to exclude signs and trailing periods in numeric constants.

To determine the effective offset of a data item three values are added together: the base offset contained in the item's symbol table entry, the displacement of the current record from the first record of the block and the displacement of this occurrence of the item from the first occurrence of the item. The base offset is the location of the location of the leftmost character of the first (that occurrence with a subscript of one) or only occurrence of the item

in the first record of a block. By using a pointer in the data item's symbol table entry, the symbol table entry of the appropriate file is reached; an entry in the file's symbol table entry contains the record displacement (if the last record read in an input file was the ith in the block and the record length is ℓ then the record displacement is $\ell(i-1)$). Items in WORKING-STORAGE always have a zero record displacement. If the item has a subscript, the third value is determined by first obtaining the value of the subscript by recursively calling "address" and calling a routine in the machine. An field in the item's symbol table entry points to the item with the OCCURS clause; the length of the latter item and the value of the subscript determine the occurrence displacement (if ℓ is the length and i the subscript value, the occurrence displacement is $\ell(i-1)$.

The other fields of an operand list entry are reformatted symbol table information or are initialized to zero. One exception applies to numeric constants. In order to avoid repeated decimal-to-binary conversions, the constant's binary value is determined once, then saved in a symbol table field (if it can be expressed in 12 unsigned bits). This value is copied by "address" into the value field of the operand list.

4.3 The machine

The machine operates in three modes: binary,

decimal and character. The mode of an instruction is a function of the operation code and the types of two operands. Because all arithmetic operations are done in binary mode, the binary mode subroutine is called directly by the supervisor for ADD, MULTIPLY and SUBTRACT statements. For expression evaluation, including comparisons, and for MOVE statements a three-dimensional array, initialized at the start of interpretation and referenced upon entry into the machine, determines the mode. The subscripts of the array reference are the types of the two operands and the type of operation code; the values in the array are 0, 1, 2 and 3 indicating illegal cases, character mode, decimal mode and binary mode, respectively.

The machine has an instruction set of 12 commands:

1 divide
2 multiply
3 add
4 subtract
5 complement algebraically
7 move
8 store
9 compare for less than
10 compare for equal to
11 compare for greater than
12 complement logically
15 test for numeric

In addition one subroutine of the machine is often called directly to return a value needed immediately by the supervisor (for example when evaluating a subscript).

When evaluating an expression the machine executes each instruction in the queue prepared by the supervisor. As mentioned earlier some of the operand list pointers may be incorrect. This is determined by checking the format of the operand pointed to. If it is null, the operand list is searched sequentially for the first operand whose format is not null; this is the correct operand. For involved expressions requiring many intermediate results this algorithm is rather inefficient. However complicated expressions should be rare in most COBOL programs. This algorithm does permit a convenient scheme for allocating intermediate results.

Although two operands are used to determine the mode, the machine executes instructions using zero, one, two and three operands depending on the instruction. For example, logical negation has no explicit operand (it uses a reserved variable) while addition has three. The sum of the first two operands is stored in the third. If zero is specified as the third operand, the machine is to establish an operand list entry for an intermediate result; it will be in one of the slots vacated by erasing the first two operands from the list. This arrangement proved to be very flexible; it was developed after investigation showed that a fixed number of operands would be cumbersome.

4.3.1 Binary mode

Binary mode is by far the most complex of the three modes. It includes all arithmetic and editing operations as well as moves and compares involving binary operands.

The central problems faced by the binary mode arithmetic units have been clearly stated by Lippit [6]:

> (1) How can we achieve the same arithmetic result on a nondecimal machine as on a decimal machine?
>
> (2) Regardless of machine type, what are the rules to be followed in performing the arithmetic operation?

Lippit's solution to the first problem is used: simulate decimal arithmetic keeping track of decimal point alignment. To do this each value is represented as integer-valued floating-point number and a power of ten. This avoids the representation errors in noninteger floating-point numbers which do not occur on a decimal machine.

The rules to be followed are partially described by COBOL options. The ROUNDED option requires that results be rounded, but because this option is not available in the subset results instead must always be chopped. Although the SIZE ERROR option is not part of the subset's syntax, it is implemented for all arithmetic operations as an aid to the student in debugging his programs. Checks for overflows causing loss of significance are made. If an overflow occurs, an error message is displayed so that the student may enter a smaller value.

The number of integer and decimal places to be carried in intermediate results is not specified by ANSI. The rules used for intermediate results are essentially those given for IBM 360 COBOL [3]. These rules specify how to carry the maximum number of significant digits in each intermediate result based on the number of integer and decimal places of each operand and on the maximum number of decimal places of any operand. The only modification made was to reduce the maximum number of significant decimal digits from 30 to 14 in order to fit any value into one CYBER 73 word. This is a mild restriction since 14 should normally be ample.

Binary mode must convert all decimal operands to binary, perform its computations in binary, then convert the result to decimal if necessary. The binary-to-decimal conversion is aided by a TUTOR command for this function, but no decimal-to-binary conversion is available. The decimal-to-binary routine verifies that each digit position does in fact contain a digit. If not an error routine displays the illegal character and accepts a correction.

Also included in binary mode is the editing of numeric edited fields. This includes suppression of leading zeros, replacement of leading zeros by asterisks, fixed and floating dollar signs and minus signs, and insertion of periods, commas, blanks and zeros, as prescribed by picture strings. Because there is no equivalent instruction in

TUTOR, a fairly large subroutine is required. Information about the picture stored by the editor in the picture's symbol table entry, as well as the characters of the picture string, dictates the progress of this subroutine.

4.3.2 Decimal mode

The move and compare operations are executed by the decimal mode routine in a direct manner. The operations proceed one character at a time from left to right in the decimal operands. Decimal point alignment is accomplished by appropriate initialization of the pointers to characters within the operands. As in the decimal-to-binary conversion each digit position must contain a digit or else an error occurs.

4.3.3 Character mode

This is the least complicated of the machine's three modes. Its operations are moves, comparisons and numeric tests. They proceed left to right one character at a time, with truncation and padding with spaces in moves and with extension of the shorter operand with spaces in comparisons. No error conditions are possible.

5. DOCUMENTATION PROVISIONS

As with almost any system of programs, the PLATO COBOL system will probably require changes in the future. Therefore documentation is important as a source of guidance to someone who needs to understand and change the COBOL tables or lessons. Hopefully a person with the proper background who carefully studies the documentation will acquire a thorough understanding of the COBOL system.

Changes may be needed for many reasons. Both PLATO and the compiler-writing system are relatively new and may be changed considerably. Changes in either might require changes in the COBOL lessons or might permit improvements in them. Other features might be added to the COBOL subset, although currently space restrictions prevent this. It is possible that there are undetected errors in the system which need to be fixed, although the system has been thoroughly tested.

In order to fully understand the COBOL system, a person needs to have a solid background in several areas. Of course he needs to understand the principles of compiler construction and of the compiler-writing system and he needs a good knowledge of the TUTOR language. In order to understand the rules of COBOL that are imposed upon the student's program and obeyed by the interpreter, he needs a detailed

knowledge of COBOL, or at least a fair knowledge plus a handy reference manual.

Documentation separate from the programs it documents can easily become obsolete as changes are made to the programs. To reduce this problem most of the documentation is included with the programs, accessible to a future author through his PLATO terminal. This should encourage the future author to change the documentation whenever he change a program. The only exception is this thesis, which is an introductory guide to the structure of the COBOL system and to the major problems encountered in it. The thesis is more general than the other documentation and less subject to change.

Two sources provide information about the COBOL subset. One description of the subset, which uses standard COBOL notation and explains many features of the COBOL system, is available to both students and authors in lesson "trw2" and is included in Appendix A. Another description using an extended BNF notation is included within lesson "cobolcomp." Also in "cobolcomp" are a guide to all documentation, an explanation of how fields of the symbol table are used for different syntactic classes, and an explanation of the modifications made to accommodate two sets of lexical tables.

The tables of the system are maintained by auxiliary programs of the compiler-writing system and may be easily inspected. Most of the COBOL tables, given in Appendix B, can be understood just by studying their entries. The lexical node table is an exception; it is very difficult to follow the lexical recognition of a token by inspecting the table. To clearly display how the node table constructs tokens, a short TUTOR lesson to generate transition diagrams based on the node tables is available; a sample is contained in Appendix D.1.

The most detailed documentation is provided by extensive comments included within the programs ("cobolcomp," the syntax program, and "cobolrun"). The comments explain the function of variables and tell the purpose of program units and of the more obscure statements. However since the comments presume a knowledge of the more general documentation, the program listings should not be the first documentation studied.

As further documentation for the interpreter two TUTOR routines provide graphic representations on a PLATO screen. One is a set of diagrams showing how the units of "cobolrun" are related (Appendix D.2); the other is a diagram displaying the table that determines the mode of the machine (Appendix D.3).

6. CONCLUSIONS

The implementation of the interactive COBOL compiler system on PLATO has demonstrated the general suitability of the compiler-writing system to COBOL. Although we have seen that COBOL is not ideally suited to the system, the system is sufficiently flexible to accommodate almost all requirements of COBOL.

While the table-driven system is available for the editor part of the compiler, the interpreter part is a program written exclusively for COBOL and required much more time than the editor. An extension to the compiler-writing system to include some form of language-independent interpretation is a large and difficult area for further research that could significantly reduce the time needed to implement a new language using this system.

The COBOL system has also demonstrated the restrictions that the current PLATO system places on large lessons requiring a relatively high usage of processor time. PLATO memory restrictions are the primary cause of the size limitations imposed on the COBOL subset and on COBOL programs. The rate of CPU usage needed by the COBOL system exceeds the PLATO average (designed for lessons with many fewer calculations) and causes slower response times than normally found on PLATO. Because the COBOL system has not yet

been used by large numbers of students, it is uncertain to what extent these problems will limit the system's usefulness.

LIST OF REFERENCES

1. Armstrong, Russell M., Modular Programming in COBOL, John Wiley and Sons, New York, 1973.

2. Gries, David, Compiler Construction for Digital Computers, John Wiley and Sons, New York, 1971.

3. IBM Corporation, IBM OS Full American National Standard COBOL, 4th edition, 1972.

4. IBM Corporation, IBM System/360 Principles of Operation, 9th edition, Poughkeepsie, New York, 1970.

5. Knuth, Donald E., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

6. Lippet, A., "COBOL and Compatibility," Communications of the Association for Computing Machines, V (May 1962), 254-255.

7. Lyman, Elisabeth R., PLATO Highlights, Computer-based Education Research Laboratory, University of Illinois, Urbana, Illinois, 1974.

8. Nievergelt, Jurg, Reingold, E. M. and Wilcox, T. R. "The Automation of Introductory Computer Science Courses," International Computing Symposium 1973, 495-501, North-Holland Publishing Co., 1974.

9. Stifle, Jack, The PLATO IV Architecture, Computer-Based Education Research Laboratory, University of Illinois, Urbana, Illinois, 1971.

10. USASI Working Group on COBOL Standards, COBOL Information Bulletin #9 [proposed USASI standard COBOL], USA Standards Institute, New York, January 1967.

11. Weinberg, Gerald M., The Psychology of Computer Programming, Van Nostrand Reinhold Company, New York, 1971.

12. Wilcox, Thomas R., "Generating machine code for high-level programming languages," Ph.D. Thesis, Department of Computer Science Report No. TR-71-103, Cornell University, Ithaca, New York, September 1971.

13. Wilcox, Thomas R., "The Interactive Compiler as a Consultant in the Computer Aided Instruction of Programming," Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, 396-399, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, 1973.

APPENDIX A

COBOL HELP LESSON

Appendices A, B, C and D consist of copies of displays seen on a PLATO screen. This appendix is the description of the COBOL subset contained in lesson "trw2" which the student can reference while writing his program in the editor.

COBOL Subset on PLATO

The following pages give a description of the COBOL subset implemented on PLATO. It is assumed that you have some knowledge of COBOL fundamentals.

Press:	for:
1	General Information
2	Format by Columns
3	Character set
4	Syntax Notation
5	DATA DIVISION
6	Print Files
7	PROCEDURE DIVISION
HELP	Explanation of Keys Used

General Information about PLATO-COBOL

This compiler is intended for use by beginning students of COBOL. A subset of COBOL is recogized which should be sufficient for this purpose.

The size of a COBOL program is restricted to the size of the PLATO screen. Since COBOL programs tend to be large, this is a significant restriction. In order to minimize the problem, some modifications to standard COBOL have been made. One is the column format, as given on the next page.

A second change is that the IDENTIFICATION DIVISION and the ENVIRONMENT DIVISION are not included. You begin your program by entering FD in columns 2 and 3.

All files must have LABEL RECORDS OMITTED. It is assumed that it has been ASSIGNed to the PLATO terminal. This means that all input must be supplied by the student through the keyboard, while output will be displayed on the screen.

Use of Columns in PLATO-COBOL

```
         1111111111222222222233333333334444444444555555555566666
1234567890123456789012345678901234567890123456789012345678901234
c1111xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

c: Column 1: same as normal column 7:
space: for most lines
\- to continue a name or literal from the previous line
* to make the entire line a comment

1111: Columns 2-5: Area A:
All DIVISION and SECTION headers, paragraph names, FDs and Ø1 and 77 must begin in these columns.

> To indicate the physical end of your program, put END in area A.

xxxxx...: Columns 6-64: Area B:
Everything else goes here.

There is no statement number field (usually columns 1-6).

Punctuation: the comma and semi-colon may not be used.

COBOL Character Set

The PLATO COBOL compiler uses the EBCDIC character set (the one used by IBM 360/370 computers and others). All of the special EBCDIC characters are available.

COLLATING SEQUENCE (low to high):

␣	space
¢	cents sign (press ⇐)
.	period
<	less than
(	left parentheses
+	plus sign
|	logical OR (press SHIFT -)
&	logical AND (press SHIFT +)
!	exclamation point
$	currency symbol
*	asterisk
)	right parentheses
;	semicolon
¬	logical NOT (press SHIFT ⇐)
-	hyphen, minus sign
/	slash
,	comma
%	percent sign
_	underscore
>	greater thar
?	question mark
:	colon
#	number sign (press SHIFT 2)
@	at (press SHIFT 3)
'	apostrophe
=	equal sign
"	quotes (use this rather than ' for nonnumeric literals)
A-Z	letters (you don't need to press SHIFT)
0-9	digits

Explanation of Symbols Used in Syntax Descriptions:

CAPS	Required keyword
CAPS	Optional word
small	User-supplied word or literal
{ }	Choice to be made
[ ]	Optional phrase
....	Optional repetition of previous entry

DATA DIVISION

Press...	for...
1	File Description Entries
2	Record Description Entries
3	Alphanumeric Data-items
4	Numeric-edited Data-items
5	Numeric Data-items
6	Other information
LAB	to get back to this page

File Descriptions

FD file-name

LABEL { RECORD IS / RECORDS ARE } OMITTED

[BLOCK CONTAINS integer { CHARACTERS / RECORDS }]

[RECORD CONTAINS integer CHARACTERS] .

--- only fixed length records are implemented .

Record Description Entries

```
level-number  {item-name}  [REDEFINES item-name]
              {FILLER  }

    [OCCURS integer TIMES]  [VALUE IS literal]

    [USAGE IS {  DISPLAY     }]
              {COMPUTATIONAL }

    [PIC picture-string] .
```

--- only 1 level of subscripting is allowed
--- see next topic for details of acceptable pictures
--- level number may not be 66 or 88

To insure that there will be enough space for your input, make sure that no elementary item in an input file is larger than 75 characters!!!

Alphanumeric Data-items

An alphanumeric data-item (AN) is an item whose picture contains only the letter X. The repetition factor may be used; this means that XXXXX and X(5) both are valid pictures for a field that may contain any 5 characters.

Numeric-Edited data-items (NE)

A numeric-edited data-item (NE) is one whose picture contains at least one of the following: $ - * Z . , Ø and B. It may also cortain: 9 and V. The symbols + P CR and DB which are part of standard COBOL numeric-edited pictures are not available on PLATO. The minus sign - may not appear at the right end of the picture.

Numeric Data-items

A numeric item is one whose picture has at least one 9. It may have S and V. P is not available.

A numeric item is assumed to be in external decimal format (ED) unless its USAGE is COMPUTATIONAL, in which case its format is binary (BI). USAGE COMP in PLATO-COBOL may only be declared in WORKING-STORAGE, and pictures for COMP items must have an S and 8 9s [for example S9(8) and S9(6)V99 are valid]. The SYNC clause may not be specified, but it is implicitly in effect for all COMP items; each COMP item is 4 characters long. Using REDEFINES together with COMP is strongly discouraged.

The maximum number of digit positions which may be specified for a ED item is 14 (normally 18 digits are allowed).

Other Info on DATA DIVISION

Alphabetic items (pictures with A) and alphanumeric-edited items (pictures with X A B , Ø) are not available.

OUTPUT "PRINT" FILES

1. All output will be displayed on your PLATO screen in the same way as a COBOL file is written to the printer. The screen has 32 lines, each line has 64 positions.

2. The record size should be 65 characters or less. This must include 1 position (the first) for a compiler-generated control character. Do not move anything to this position; it won't be shown if you do.

3. A block of a "print" file is considered by this compiler to be a "page"; that is, the amount of print output to be displayed on the screen at one time. This means that if the BLOCK CONTAINS clause is omitted or is 1 RECORDS, the output will be shown one line at a time. If, for example, BLOCK CONTAINS 15 RECORDS, no output will be shown until the 15th WRITE. Because the maximum number of character that can be stored is 1792, the absolute maximum number of 64 character lines/page is 27 (this would leave only 37 characters for all other data).

Press...	for...
1	ADD, CLOSE, COMPUTE, EXIT, GO, IF, MOVE, MULTIPLY
2	OPEN, PERFORM, READ, STOP, SUBTRACT, WRITE
3	Legal MOVEs
4	Legal Comparisons
5	Types of Conditions Available
6	Other Information
LAB	To return to this page

ADD {data-item / literal} .... TO data-item

ADD {data-item / literal} {data-item / literal} .... GIVING data-item

CLOSE file-name ....

COMPUTE data-item = arithmetic-expression

EXIT

GO TO paragraph-name

GO TO paragraph-name.... DEPENDING ON data-item

IF condition {NEXT SENTENCE / statement...} [ELSE {NEXT SENTENCE / statement....}]

MOVE {data-item / literal} TO data-item ....

MULTIPLY {data-item / literal} BY data-item

MULTIPLY {data-item / literal} BY {data-item / literal} GIVING data-item

```
OPEN {INPUT } file-name...
     {OUTPUT}

PERFORM paragraph-name [THRU paragraph-name]
        [{data-item} TIMES]
         { integer }

PERFORM paragraph-name [THRU paragraph-name]
      [VARYING data-item FROM {data-item} BY {data-item}]
                             { literal }    { integer }
      UNTIL condition

READ file-name AT END statement....

STOP RUN

SUBTRACT {data-item}.... FROM data-item
         { literal }

SUBTRACT {data-item}.... FROM {data-item} GIVING data-item
         { literal }          { literal }

WRITE record-name [BEFORE ADVANCING integer LINES]
```

C = nonnumeric move N = numeric move

X = illegal move 1 = numeric item must be an integer

FROM \ TO	Group item	Alpha-numeric	Numeric edited	Numeric DISPLAY	Numeric COMP
Group item	C	C	C	C	C
Alphanumeric item	C	C	N	N	X
Numeric Edited item	C	C	X	X	X
SPACES or Nonnumeric literal	C	C	X	X	X
ZERO	C	C	N	N	N
Numeric literal	X	X	N	N	N
Numeric DISPLAY item	C	C[1]	N	N	N
Numeric COMP item	C	N	N	N	N

	a	b	c	d	e	f
a	C	C	C	C	X	X
b	C	X	X	C	X	X
c	C	X	X	N	X	N
d	C	C	N	N	N	N
e	X	X	X	N	X	N
f	X	X	N	N	N	N

C = nonnumeric compare

N = numeric compare

X = illegal compare

a: group or alphanumeric or numeric edited item
b: SPACES or nonnumeric literal
c: ZERO or integer numeric literal
d: numeric DISPLAY integer
e: fractional numeric literal
f: numeric DISPLAY fraction or COMP item or expression

Information about Conditions

The only types of tests allowed in conditions are the relation test (< = >) and the test for NUMERIC. Other class tests, sign tests and condition-name tests are not provided.

Implied subjects or relational operators (such as "IF A = B OR C...") are not allowed.

Compound conditions may be formed by connecting simple conditions with AND or OR, but AND and OR cannot be used in the same compound condition. NOT cannot be used as a logical operator, but may be used in relation tests. This means that IF A NOT = B is ok, but IF NOT A = B will cause an error message.

Other Info about the PROCEDURE DIVISION

The ROUNDED option is not available; no rounding will occur. Although SIZE ERROR cannot be specified, it is in effect at all times; you will get warnings at exection time if arithmetic overflows occur.

Full expressions may be written in COMPUTE statements and in conditions, except that exponentiation (**) is not provided.

Each sentence, including the first one, must be in a paragraph. Section names can't be used, however.

At the end of your program, you must put END in area A (columns 2-5) followed by a space, then press STOP.

APPENDIX B

COBOL TABLES IN THE COMPILER-WRITING SYSTEM

All COBOL tables used by the language-independent editor routines, except the syntax code table, are included in this appendix. The displays shown here are not seen by the student writing a COBOL program but by the COBOL implementor when constructing or modifying the tables.

B.1 Primary tables

Press...	To change...
a	Name of langauge (COBOL)
b	Name of character set (COBOL in trw2)
c	Editor leslist index <15>: (cobolcomp)
d	Interpreter leslist index <20>: (cobolrun)
e	Internal character codes (64 codes)
f	Blank code (1)
g	Default tabs
h	Key Function Tables
i	Lexical Class Assignments (20 classes)
j	Lexical Node Table (30 nodes)
k	Safe Nodes
l	Predefined Symbols (110 names, 565 characters)
m	Field Symbols
n	New Symbol Allocation Classes

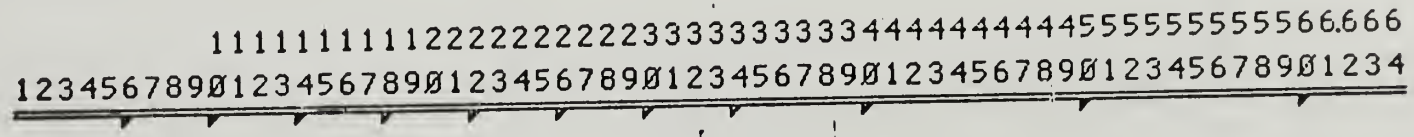
SHIFT BACK to return tables to -trw6-

B.1.1 Internal character codes

Internal Character Codes

	16- /	32- E	48- U	64- ↑
1- .	17- ,	33- F	49- V	
2- ¢	18- %	34- G	5Ø- W	
3- .	19- _	35- H	51- X	
4- <	2Ø- >	36- I	52- Y	
5- (	21- ?	37- J	53- Z	
6- +	22- :	38- K	54- Ø	
7- |	23- #	39- L	55- 1	
8- &	24- @	4Ø- M	56- 2	
9- !	25- '	41- N	57- 3	
1Ø- $	26- =	42- O	58- 4	
11- *	27- "	43- P	59- 5	
12-)	28- A	44- Q	6Ø- 6	
13- ;	29- B	45- R	61- 7	
14- ¬	3Ø- C	46- S	62- 8	
15- -	31- D	47- T	63- 9	

Initial Tab Settings (⊽=tabset)



Enter a column number here to change the setting »
Press BACK to return to the index page.

B.1.3 Key function tables

Press...	For...
1	Lower case keyboard
2	Upper case keyboard
3	Function keys
4	Lower case access keyboard
5	Upper case access keyboard
6	Access function keys

Press BACK to return to index page
Press HELP to locate a table by key
Press SHIFT HELP for list of special function codes

-1	Ignore key
-2	Move cursor 1 character to right.
-3	Move cursor right one token
-4	Move cursor right one line
-5	Erase character to right
-6	Move cursor left one character
-7	Move cursor left one token
-8	Move cursor left one line
-9	Erase token to right
-10	Erase character to left of cursor
-11	Erase token to left of cursor
-12	Erase line to left of cursor
-13	Erase line to right
-14	Access alternate key set
-15	Tab
-16	Open "gap" one line.
-17	Clear screen
-18	Return to index page
-19	Close "gap" by one line
-20	Help sequence 1
-21	Help sequence 2
-22	Switch lesson to backgound
-23	Compiler debugging aids
-24	Move cursor to end of text
-25	Unassigned
-26	Switch lesson to foreground
-27	Set tab
-28	Clear tab
-29	Reprint screen
-30	Move cursor to next line

Lower case — Enter new (numeric) function code at arrow.

			16- p=	43	(P)	32- 5=	59	(5)	48- ÷=	16	(/)
1- a=	28	(A)	17- q=»			33- 6=	6Ø	(6)	49- [=	23	(#)
2- b=	29	(B)	18- r=	45	(R)	34- 7=	61	(7)	5Ø-]=	24	(@)
3- c=	3Ø	(C)	19- s=	46	(S)	35- 8=	62	(8)	51- %=	18	(%)
4- d=	31	(D)	2Ø- t=	47	(T)	36- 9=	63	(9)	52- ×=	11	(*)
5- e=	32	(E)	21- u=	48	(U)	37- +=	6	(+)	53- ←=	2	(↑)
6- f=	33	(F)	22- v=	49	(V)	38- -=	15	(-)	54- =	-16	
7- g=	34	(G)	23- w=	5Ø	(W)	39- *=	11	(*)	55- =	-19	
8- h=	35	(H)	24- x=	51	(X)	4Ø- /=	16	(/)	56- =	14	(¬)
9- i=	36	(I)	25- y=	52	(Y)	41- (=	5	(()	57- =	-3Ø	
1Ø- j=	37	(J)	26- z=	53	(Z)	42-)=	12	())	58- <=	4	(<)
11- k=	38	(K)	27- Ø=	54	(Ø)	43- $=	1Ø	($)	59- >=	2Ø	(>)
12- l=	39	(L)	28- 1=	55	(1)	44- ==	26	(=)	6Ø- =	-1	
13- m=	4Ø	(M)	29- 2=	56	(2)	45- =	1	(.)	61- =	Ø	
14- n=	41	(N)	3Ø- 3=	57	(3)	46- ,=	17	(,)	62- =	Ø	
15- o=	42	(O)	31- 4=	58	(4)	47- .=	3	(.)	63- ;=	13	(;)

Internal Character Codes

1-	8-&	15--	22-:	29-B	36-I	43-P	5Ø-W	57-3	64-?
2-¢	9-!	16-/	23-#	3Ø-C	37-J	44-Q	51-X	58-4	
3-.	1Ø-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-'	32-E	39-L	46-S	53-Z	6Ø-6	
5-(	12-)	19-_	26-=	33-F	4Ø-M	47-T	54-Ø	61-7	
6-+	13-;	2Ø->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

UPPER CASE Enter new (numeric) function code at arrow.

64-	=	Ø		8Ø-	P=	43	(P)	96-	=	Ø		112-	n=	21 (?)
65-	A=	28	(A)	81-	Q=	44	(Q)	97-	_=	19	(_)	113-	=	Ø
66-	B=	29	(B)	82-	R=	45	(R)	98-	'=	25	(')	114-	=	Ø
67-	C=	3Ø	(C)	83-	S=	46	(S)	99-	=	Ø		115-	=	Ø
68-	D=	31	(D)	84-	T=	47	(T)	1ØØ-	=	Ø		116-	u=	21 (?)
69-	E=	32	(E)	85-	U=	48	(U)	1Ø1-	Σ=	8	(&)	117-	=	Ø .
7Ø-	F=⟫			86-	V=	49	(V)	1Ø2-	Δ=	7	(I)	118-	=	Ø '
71-	G=	34	(G)	87-	W=	5Ø	(W)	1Ø3-	=	Ø		119-	=	Ø
72-	H=	35	(H)	88-	X=	51	(X)	1Ø4-	?=	21	(?)	12Ø-	=	Ø
73-	I=	36	(I)	89-	Y=	52	(Y)	1Ø5-	=	Ø		121-	=	Ø
74-	J=	37	(J)	9Ø-	Z=	53	(Z)	1Ø6-	=	Ø		122-	=	Ø
75-	K=	38	(K)	91-	=	Ø		1Ø7-	=	Ø		123-	=	Ø
76-	L=	39	(L)	92-	=	Ø		1Ø8-	=	Ø		124-	=	Ø
77-	M=	4Ø	(M)	93-	=	Ø		1Ø9-	=	Ø		125-	=	Ø
78-	N=	41	(N)	94-	=	Ø		11Ø-	"=	27	(")	126-	=	Ø
79-	O=	42	(O)	95-	=	Ø		111-	!=	9	(!)	127-	:=	22 (:)

Internal Character Codes

1-.	8-&	15--	22-:	29-B	36-I	43-P	5Ø-W	57-3	64-?
2-¢	9-!	16-/	23-#	3Ø-C	37-J	44-Q	51-X	58-4	.
3-.	1Ø-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-'	32-E	39-L	46-S	53-Z	6Ø-6	
5-(	12-)	19-_	26-=	33-F	4Ø-M	47-T	54-Ø	61-7	
6-+	13-;	2Ø->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

[illegible] FUNCTION Enter new (numeric) function code at arrow.

```
128- x          =>
129- x          =   Ø
13Ø- next       =  -2
131- next1      =  -3
132- erase      = -1Ø
133- erase1     = -11
134- help       = -2Ø
135- help1      = -2Ø
136- back       =  -6
137- back1      =  -7
138- lab        =  -1
139- lab1       = -17
14Ø- data       =  -1
141- data1      = -23
142- term       =  -1
143- ans        =  -1
144- copy       =  -1
145- copy1      = -29
146- edit       =  -5
147- edit1      =  -9
148- micro      = -14
149- micro1     = -14
15Ø- stop       = -18
151- stop1      =  -1
152- tab        = -15
```

Internal Character Codes

1-.	8-&	15--	22-:	29-B	36-I	43-P	5Ø-W	57-3	64-??
2-¢	9-!	16-/	23-#	3Ø-C	37-J	44-Q	51-X	58-4	
3-.	1Ø-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-!	32-E	39-L	46-S	53-Z	6Ø-6	
5-(	12-)	19-_	26-=	33-F	4Ø-M	47-T	54-Ø	61-7	
6-+	13-;	2Ø->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

LOWER CASE ACCESS Enter new (numeric) function code at arrow.

1- α= Ø	16- π= Ø	32- @= Ø	48- ÷= Ø
2- β= -22	17- ` = Ø	33- ⩾= Ø	49- {= Ø
3- ´ = Ø	18- ρ= Ø	34- 7= Ø	50- }= Ø
4- δ= Ø	19- σ= Ø	35- 8= Ø	51- %= Ø
5- ´ = Ø	20- θ= Ø	36- 9= Ø	52- ×= Ø
6- f= -26	21- ¨ = Ø	37- &= Ø	53- = Ø
7- g= Ø	22- l= Ø	38- -= Ø	54- = Ø
8- h= Ø	23- ω= Ø	39- *= Ø	55- = Ø
9- i= Ø	24- ^ = Ø	40- \= Ø	56- = Ø
10- j=⩾	25- = Ø	41- (= Ø	57- = -28
11- k= Ø	26- = Ø	42- ≡= Ø	58- ≤= Ø
12- λ= Ø	27- (= Ø	43- #= Ø	59- ≥= Ø
13- μ= Ø	28-)= Ø	44- ≠= Ø	60- = Ø
14- ~ = Ø	29- 2= Ø	45- = Ø	61- = Ø
15- °= Ø	30- 3= Ø	46- ↓= Ø	62- = -14
	31- 4= Ø	47- .= Ø	63- ~= Ø

Internal Character Codes

1-.	8-&	15--	22-:	29-B	36-I	43-P	50-W	57-3	64-?
2-¢	9-!	16-/	23-#	30-C	37-J	44-Q	51-X	58-4	
3-.	10-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-'	32-E	39-L	46-S	53-Z	60-6	
5-(	12-)	19-_	26-=	33-F	40-M	47-T	54-Ø	61-7	
6-+	13-;	20->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

OTHERWISE PROCESS Enter new (numeric) function code at arrow.

64-	=»	Ø	8Ø-	P=	Ø	96-	=	Ø	112-	∩=	Ø
65-	←=	Ø	81-	Q=	Ø	97-	_=	Ø	113-	=	Ø
66-	B=	Ø	82-	R=	Ø	98-	'=	Ø	114-	=	Ø
67-	C=	Ø	83-	S=	Ø	99-	=	Ø	115-	=	Ø
68-	→=	Ø	84-	T=	Ø	1ØØ-	=	Ø	116-	=	Ø
69-	E=	Ø	85-	U=	Ø	1Ø1-	Σ=	Ø	117-	=	Ø
7Ø-	F=	Ø	86-	V=	Ø	1Ø2-	Δ=	Ø	118-	=	Ø
71-	G=	Ø	87-	↑=	Ø	1Ø3-	=	Ø	119-	=	Ø
72-	H=	Ø	88-	↓=	Ø	1Ø4-	?=	Ø	12Ø-	=	Ø
73-	I=	Ø	89-	Y=	Ø	1Ø5-	=	Ø	121-	=	Ø
74-	J=	Ø	9Ø-	Z=	Ø	1Ø6-	=	Ø	122-	=	Ø
75-	K=	Ø	91-	=	Ø	1Ø7-	=	Ø	123-	=	Ø
76-	L=	Ø	92-	=	Ø	1Ø8-	=	Ø	124-	=	Ø
77-	M=	Ø	93-	=	Ø	1Ø9-	=	Ø	125-	=	Ø
78-	N=	Ø	94-	=	Ø	11Ø-	"=	Ø	126-	=	Ø
79-	O=	Ø	95-	=	Ø	111-	!=	Ø	127-	:=	Ø

Internal Character Codes

1-.	8-&	15--	22-:	29-B	36-I	43-P	5Ø-W	57-3	64-?
2-¢	9-!	16-/	23-#	3Ø-C	37-J	44-Q	51-X	58-4	
3-.	1Ø-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-'	32-E	39-L	46-S	53-Z	6Ø-6	
5-(	12-)	19-_	26-=	33-F	4Ø-M	47-T	54-Ø	61-7	
6-+	13-;	2Ø->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

ACCESS FUNCTION Enter new (numeric) function code at arrow.

```
128- access-x   =»              144- access-copy=  -1
129- access-x   =  Ø            145- access-copy=  -1
13Ø- access-next= -4            146- access-edit= -13
131- access-next= -24           147- access-edit=  -1
132- access-eras= -12           148- access-micr=  -1
133- access-eras=  -1           149- access-micr=  -1
134- access-help=  -1           15Ø- access-stop=  -1
135- access-help=  -1           151- access-stop=  -1
136- access-back=  -8           152- access-tab = -27
137- access-back= -25
138- access-lab =  -1
139- access-lab1=  -1
14Ø- access-data=  -1
141- access-data=  -1
142- access-term=  -1
143- access-ans =  -1
```

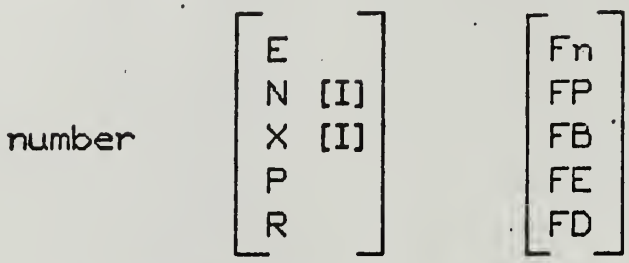
Internal Character Codes

1-.	8-&	15--	22-:	29-B	36-I	43-P	5Ø-W	57-3	64-?
2-¢	9-!	16-/	23-#	3Ø-C	37-J	44-Q	51-X	58-4	
3-.	1Ø-$	17-,	24-@	31-D	38-K	45-R	52-Y	59-5	
4-<	11-*	18-%	25-'	32-E	39-L	46-S	53-Z	6Ø-6	
5-(	12-)	19-_	26-=	33-F	4Ø-M	47-T	54-Ø	61-7	
6-+	13-;	2Ø->	27-"	34-G	41-N	48-U	55-1	62-8	
7-|	14-¬	21-?	28-A	35-H	42-O	49-V	56-2	63-9	

```
Class Members....
 none - ?
    1 - A B D E F G H J K L M N O Q R S T U V W X Y Z
    2 - Ø 1 2 3 4 5 6 7 8 9
    3 - .
    4 - "
    5 - .
    6 - (
    7 - )
    8 - < > =
    9 - -
   1Ø - +
   11 - *
   12 - /
   13 - ¢ | & ! $ ¬ % _ ? : # @ '
   14 - ; ,
   15 - C1
   16 - C2 C3 C4 C5
   17 - C6 C7 C8 C9 C1Ø C11 C12 C13 C14 C15 C16 C17 C18 C19 C2Ø
        C21 C22 C23 C24 C25 C26 C27 C28 C29 C3Ø C31 C32 C33 C34 C
        35 C36 C37 C38 C39 C4Ø C41 C42 C43 C44 C45 C46 C47 C48 C4
        9 C5Ø C51 C52 C53 C54 C55 C56 C57 C58 C59 C6Ø C61 C62 C63
         C64
   18 - P
   19 - I
   2Ø - C
```

B.1.5 Lexical node table

Each table entry has the form: ([]=optional)



The letter codes have the following meaning:

Letter	Meaning....	Interpretation of number
E	Lexical error	error message number
N	Non-extendible token	syntactic class
X	eXtendible token	syntactic class
I	Ignore token (do not send to parser)	N. A.
P	Procedure call	first node of procedure
R	Return from procdure	return code
(blank)	Accept character	next node

Fn	Generate field #n (1, 2, or 3)	
FP	Field starts with next column	Continuation field
FB	Field begins with current column	
FE	Field ends with current column	
FD	Field ends with previous column	

Press NEXT to inspect the node table.
Press BACK to return to the index page.

NODE ↓↓	CHARACTER CLASSES→ 1 (A)	2 (Ø)	3 (.)	4 (")	5 (.)
Ø	1	3	27	1Ø	8
1	1	1	21 X	1 E	21 X
2	1	1	2 E	2 E	2 E
3	1	3	4 X	1 E	4
4	3 E	7	8 X	3 E	3 E
5	3 E	13	1Ø N	3 E	6
6	4 E	7	4 E	4 E	4 E
7	5 E	7	5 X	5 E	5 X
8	3 E	7	14 N	3 E	3 E
9	1 XI	1 XI	9	1 XI	1 XI
1Ø	1Ø	1Ø	1Ø	11	1Ø
11	6 E	6 E	7 X	6 E	7 X
12	8 E	8 E	1Ø N	8 E	8 E
13	5 E	13	4 X	5 E	4
14	12 X	12 X	61 E	12 X	12 X
15	8 E	8 E	11 N	8 E	8 E
16	6 E	6 E	13 N	6 E	13 X
17	7 E	7 E	9	7 E	7 E
18	18	18	18	18	18
19	1 F1	3 F1	9	9 E	23
2Ø	1Ø E	1Ø E	1Ø E	1Ø E	1Ø E
21	8 E	8 E	21	8 E	8 E
22	11 E	11 E	22	1Ø FE	11 E
23	8 E	8 E	14 N	8 E	8 E
24	1	1	21 X	1 E	21 X
25	1	1	21 X	1 E	21 X
26	1	1	Ø F2	1 E	21 X
27	1 XI	1 XI	9	1 XI	1 XI
28	15 E	15 E	3 R	15 E	15 E
29	8 E	8 E	29	8 E	8 E
3Ø	1 R FD	2 R FD	3Ø	4 R FD	5 R FD

NODE ↓↓	CHARACTER CLASSES→ 6 (()	7 ())	8 (<)	9 (-)	10 (+)
Ø	14	16	15	5	5
1	8 E	21 X	1 E	2	1 E
2	2 E	2 E	2 E	2	2 E
3	8 E	4 X	1 E	2	1 E
4	3 E	3 E	3 E	3 E	3 E
5	3 E	3 E	3 E	3 E	3 E
6	4 E	4 E	4 E	4 E	4 E
7	5 E	5 X	5 E	5 E	5 E
8	3 E	3 E	3 E	3 E	3 E
9	1 XI	58 E	1 XI	1 XI	1 XI
10	10	10	10	10	10
11	6 E	7 X	6 E	6 E	6 E
12	8 E	8 E	8 E	8 E	8 E
13	5 E	4 X	5 E	5 E	5 E
14	12 X	62 E	60 E	12 X	12 X
15	8 E	8 E	8 E	8 E	8 E
16	6 E	13 X	6 E	6 E	6 E
17	7 E	7 E	7 E	7 E	7 E
18	18	18	18	18	18
19	9 E	9 E	9 E	9 E	9 E
20	10 E	10 E	10 E	21 FB	10 E
21	8 E	8 E	8 E	8 E	8 E
22	11 E	11 E	11 E	11 E	11 E
23	8 E	8 E	8 E	8 E	8 E
24	8 E	21 X	1 E	2	1 E
25	8 E	21 X	1 E	2	1 E
26	8 E	21 X	1 E	2	1 E
27	14	58 E	1 XI	1 XI	1 XI
28	15 E	15 E	15 E	29 FB	15 E
29	8 E	8 E	8 E	8 E	8 E
30	6 R FD	7 R FD	8 R FD	9 R FD	10 R FD

NODE ↓↓	CHARACTER CLASSES→ 11 (*)	12 (/)	13 (¢)	14 (;)	15 ()
Ø	12	15	57 E	56 E	17
1	1 E	1 E	57 E	56 E	28 P
2	2 E	2 E	57 E	2 E	28 P
3	1 E	1 E	57 E	56 E	28 P
4	3 E	3 E	57 E	3 E	28 P
5	3 E	3 E	57 E	3 E	28 P
6	4 E	4 E	57 E	4 E	28 P
7	5 E	5 E	57 E	56 E	28 P
8	3 E	3 E	57 E	3 E	28 P
9	1 XI	1 XI	57 E	56 E	9
1Ø	1Ø	1Ø	1Ø	1Ø	2Ø
11	6 E	6 E	57 E	56 E	28 P
12	56 E	8 E	57 E	8 E	28 P
13	5 E	5 E	57 E	5 E	28 P
14	6Ø E	6Ø E	57 E	61 E	28 P
15	8 E	8 E	57 E	8 E	28 P
16	6 E	6 E	57 E	56 E	28 P
17	18	7 E	7 E	7 E	59 E
18	18	18	18	18	1 XI
19	9 E	9 E	57 E	9 E	59 E
2Ø	1Ø E	1Ø E	1Ø E	1Ø E	59 E
21	8 E	8 E	8 E	8 E	59 E
22	11 E	11 E	11 E	11 E	63 E
23	8 E	8 E	57 E	8 E	28 P
24	1 E	1 E	57 E	56 E	28 P
25	1 E	1 E	57 E	56 E	28 P
26	1 E	1 E	57 E	56 E	28 P
27	1 XI	1 XI	57 E	56 E	9
28	3 R	15 E	15 E	15 E	59 E
29	8 E	8 E	8 E	8 E	59 E
3Ø	11 R	12 R	57 E	14 R	63 E

NODE ↓↓	CHARACTER CLASSES→ 16	(17	(18 (P)	19 (I)	20 (C)	
0	19		0	24	1	1
1	1		1	1	1	1
2	2		2	1	1	1
3	3		3	1	1	1
4	59 E		4	3 E	3 E	3 E
5	59 E		5	3 E	3 E	3 E
6	59 E		6	4 E	4 E	4 E
7	59 E		7	5 E	5 E	5 E
8	59 E		8	3 E	3 E	3 E
9	9		9	1 XI	1 XI	1 XI
10	59 E		10	10	10	10
11	59 E		11	6 E	6 E	6 E
12	59 E		12	8 E	8 E	8 E
13	59 E		13	5 E	5 E	5 E
14	59 E		14	12 X	12 X	12 X
15	59 E		15	8 E	8 E	8 E
16	59 E		16	6 E	6 E	6 E
17	59 E		59 E	7 E	7 E	7 E
18	18		18	18	18	18
19	59 E		59 E	1 F1	1 F1	1 F1
20	59 E		59 E	10 E	10 E	10 E
21	21		22	8 E	8 E	8 E
22	59 E		22	11 E	11 E	11 E
23	23		23	8 E	8 E	8 E
24	59 E		24	1	25	1
25	59 E		25	1	1	26
26	59 E		26	1	1	1
27	9		27	1 XI	1 XI	1 XI
28	59 E		59 E	15 E	15 E	15 E
29	29		30	8 E	8 E	8 E
30	59 E		30	18 R FD	19 R FD	20 R FD

The following nodes are currently designated as "safe" nodes:

Ø 1 2 3 9 18 27

Press [EDIT] to add nodes to this list.
[ERASE] to delete nodes from this list
[BACK] to return to index page
[HELP] to for an explanation of "safe" nodes

When the user types a CR the editor must know whether to pad the line with blanks or cut it short. To determine this the editor feeds blanks to Lexi until it enters a node which been designated as a safe place to terminate the line. Typical "safe" nodes would be the initial node and the node used to collect a string of blanks.

B.1.7 Predefined symbols

```
110 Names    565 Characters
ntp Name                stp  typ  idn  blk  off  len  dvl
  1 IDENTIFICATION        1   30    1    0    0    0    0
  2 ENVIRONMENT           2   30    2    0    0    0    0
  3 DATA                  3   30    3    0    0    0    0
  4 PROCEDURE             4   23    4    0    0    0    0
  5 FILE                  5   30    8    0    0    0    0
  6 FD                    6   23    9    0    0    0    0
  7 WORKING-STORAGE       7   23   10    0    0    0    0
  8 AREA A                8   19   15    0   15   15    5
  9 .                     9   14   16    0    0    0    0
 10 DIVISION             10   25   31    0    0    0    0
 11 SECTION              11   25   32    0    0    0    0
 12 SELECT               12   30   33    0    0    0    0
 13 BLOCK                13   25   36    0    0    0    0
 14 LABEL                14   25   37    0    0    0    0
 15 RECORD               15   25   38    0    0    0    0
 16 REDEFINES            16   25   41    0    0    0    0
```

110 Names 565 Characters

ntp	Name	stp	typ	idn	blk	off	len	dvl
17	USAGE	17	24	44	Ø	Ø	Ø	Ø
18	VALUE	18	24	45	Ø	Ø	Ø	Ø
19	DISPLAY	19	24	46	Ø	Ø	Ø	Ø
2Ø	COMP	2Ø	24	47	Ø	Ø	Ø	Ø
21	COMPUTATIONAL	2Ø	24	47	Ø	Ø	Ø	Ø
22	OCCURS	21	24	48	Ø	Ø	Ø	Ø
23	PIC	22	24	49	Ø	Ø	Ø	Ø
24	PICTURE	23	3Ø	5Ø	Ø	Ø	Ø	Ø
25	ARE	24	25	51	Ø	Ø	Ø	Ø
26	ASSIGN	25	3Ø	52	Ø	Ø	Ø	Ø
27	CHARACTERS	26	25	53	Ø	Ø	Ø	Ø
28	CONTAINS	27	25	54	Ø	Ø	Ø	Ø
29	FILLER	28	25	55	Ø	Ø	Ø	Ø
3Ø	ID	29	3Ø	56	Ø	Ø	Ø	Ø
31	IS	3Ø	25	57	Ø	Ø	Ø	Ø
32	OF	31	3Ø	58	Ø	Ø	Ø	Ø

110 Names 565 Characters

ntp	Name	stp	typ	idn	blk	off	len	dvl
33	OMITTED	32	25	59	Ø	Ø	Ø	Ø
34	RECORDS	33	25	6Ø	Ø	Ø	Ø	Ø
35	STANDARD	34	3Ø	61	Ø	Ø	Ø	Ø
36	THRU	35	25	62	Ø	Ø	Ø	Ø
37	TIMES	36	25	63	Ø	Ø	Ø	Ø
38	TO	37	25	64	Ø	Ø	Ø	Ø
39	VALUES	38	3Ø	65	Ø	Ø	Ø	Ø
4Ø	IN	39	3Ø	66	Ø	Ø	Ø	Ø
41	ON	4Ø	25	67	Ø	Ø	Ø	Ø
42	BEFORE	41	25	68	Ø	Ø	Ø	Ø
43	AFTER	42	3Ø	69	Ø	Ø	Ø	Ø
44	AT	43	25	7Ø	Ø	Ø	Ø	Ø
45	BY	44	25	71	Ø	Ø	Ø	Ø
46	DEPENDING	45	25	72	Ø	Ø	Ø	Ø
47	FROM	46	25	73	Ø	Ø	Ø	Ø
48	GIVING	47	25	74	Ø	Ø	Ø	Ø

110 Names 565 Characters

ntp	Name	stp	typ	idn	blk	off	len	dvl
49	INPUT	48	32	75	Ø	Ø	Ø	Ø
5Ø	INTO	49	3Ø	76	Ø	Ø	Ø	Ø
51	OUTPUT	5Ø	32	77	Ø	Ø	Ø	Ø
52	RUN	51	25	78	Ø	Ø	Ø	Ø
53	SENTENCE	52	25	79	Ø	Ø	Ø	Ø
54	UNTIL	53	25	8Ø	Ø	Ø	Ø	Ø
55	VARYING	54	25	81	Ø	Ø	Ø	Ø
56	THAN	55	25	82	Ø	Ø	Ø	Ø
57	ADVANCING	56	25	83	Ø	Ø	Ø	Ø
58	LINES	57	25	84	Ø	Ø	Ø	Ø
59	LINE	58	3Ø	85	Ø	Ø	Ø	Ø
6Ø	SPACE	59	7	91	Ø	Ø	Ø	Ø
61	SPACES	59	7	91	Ø	Ø	Ø	Ø
62	ZERO	6Ø	6	92	Ø	Ø	Ø	Ø
63	ZEROS	61	6	93	Ø	Ø	Ø	Ø
64	ZEROES	61	6	93	Ø	Ø	Ø	Ø

110 Names 565 Characters

ntp	Name	stp	typ	idn	blk	off	len	dvl
65	ALL	65	30	99	0	0	0	0
66	.(	100	12	171	0	2	16	13
67	JUSTIFIED	65	30	99	0	0	0	0
68	ALTER	66	18	100	0	0	0	0
69	ACCEPT	66	18	100	0	0	0	0
70	DIVIDE	66	18	100	0	0	0	0
71	EXAMINE	66	18	100	0	0	0	0
72	NOTE	66	18	100	0	0	0	0
73	ADD	69	17	104	0	0	0	3
74	CLOSE	70	17	106	0	0	0	0
75	IF	71	17	108	0	0	0	0
76	END	72	25	109	0	0	0	0
77	COMPUTE	73	17	112	0	2	16	8
78	ELSE	74	17	114	0	0	0	0
79	MOVE	75	17	116	0	0	0	7
80	MULTIPLY	76	17	117	0	0	0	2

110 Names 565 Characters

ntp	Name	stp	typ	idn	blk	off	len	dvl
81	EXIT	77	17	118	Ø	Ø	Ø	Ø
82	GO	78	17	12Ø	Ø	Ø	Ø	Ø
83	OPEN	79	17	122	Ø	Ø	Ø	Ø
84	NEXT	8Ø	25	123	Ø	Ø	Ø	Ø
85	PERFORM	81	17	124	Ø	Ø	Ø	Ø
86	READ	82	17	126	Ø	Ø	Ø	Ø
87	STOP	83	17	127	Ø	Ø	Ø	Ø
88	SUBTRACT	84	17	128	Ø	Ø	Ø	4
89	WRITE	85	17	13Ø	Ø	Ø	Ø	Ø
9Ø	AND	86	15	141	Ø	Ø	Ø	Ø
91	OR	87	15	142	Ø	Ø	Ø	Ø
92	NOT	88	15	15Ø	Ø	6	5	12
93	= .	89	11	151	Ø	8	7	1Ø
94	< .	9Ø	11	152	Ø	8	7	9
95	> .	91	11	153	Ø	8	7	11
96	LESS	92	11	154	Ø	8	7	9

```
110 Names     565 Characters
ntp Name                    stp   typ   idn   blk   off   len   dvl
 97 GREATER                  93    11   155     Ø     8     7    11
 98 EQUAL                    94    11   156     Ø     8     7    1Ø
 99 +.                       95    1Ø   161     Ø    1Ø     9     3
1ØØ -.                       96    1Ø   162     Ø    1Ø     9     4
1Ø1 /.                       97    1Ø   163     Ø    12    11     1
1Ø2 *.                       98    1Ø   164     Ø    12    11     2
1Ø3 )                        99    13   166     Ø     Ø     1    14
1Ø4 (                       1ØØ    12   171     Ø     2    16    13
1Ø5 POSITIVE                1Ø1    3Ø   173     Ø     Ø     Ø     Ø
1Ø6 NEGATIVE                1Ø2    3Ø   174     Ø     Ø     Ø     Ø
1Ø7 NUMERIC                 1Ø3    11   175     Ø     8     7    15
1Ø8 ALPHABETIC              1Ø4    3Ø   176     Ø     Ø     Ø     Ø
1Ø9 BLANK                    65    3Ø    99     Ø     Ø     Ø     Ø
11Ø ).                       99    13   166     Ø     Ø     1    14
```

B.1.8 Field tokens

Fields are special predefined, non-printable tokens which may be used to pass position information to syna:

Field	Name Table Index
1	8
2	23
3	Ø

Press '1', '2', or '3' to change any of these
Press BACK to return to index page.

New Symbol Allocation Classes

At the time a symbol is entered into the symbol table it is linked to a program block. This table indicates which block a symbol is to be added to based on the syntactic class of the symbol. The following 'block codes' are used:

Ø - Do not lirk to any block,
1 - Link to block Ø (block containing all program blocks)
2 - Link to current block,
3 - Link to block containing current block.

Here is the current table:

		8-	Ø	16-	Ø	24-	Ø	32-	Ø	4Ø-	Ø	48-	Ø	56-	Ø
1-	Ø	9-	Ø	17-	Ø	25-	Ø	33-	Ø	41-	Ø	49-	Ø	57-	Ø
2-	Ø	1Ø-	Ø	18-	Ø	26-	Ø	34-	Ø	42-	Ø	5Ø-	Ø	58-	Ø
3-	Ø	11-	Ø	19-	Ø	27-	Ø	35-	Ø	43-	Ø	51-	Ø	59-	Ø
4-	Ø	12-	Ø	2Ø-	Ø	28-	Ø	36-	Ø	44-	Ø	52-	Ø	6Ø-	Ø
5-	Ø	13-	Ø	21-	Ø	29-	Ø	37-	Ø	45-	Ø	53-	Ø	61-	Ø
6-	Ø	14-	Ø	22-	Ø	3Ø-	Ø	38-	Ø	46-	Ø	54-	Ø	62-	Ø
7-	Ø	15-	Ø	23-	Ø	31-	Ø	39-	Ø	47-	Ø	55-	Ø	63-	Ø

Type a class number to change its 'block code'>

B.2 Picture tables

Press...	To change...
a	Name of langauge (COBOL)
b	Name of character set (COBOL in trw2)
c	Editor leslist index <15>: (cobolcomp)
d	Interpreter leslist index <3>: (trw2)
e	Internal character codes (64 codes)
f	Blank code (1)
g	Default tabs
h	Key Function Tables
i	Lexical Class Assignments (20 classes)
j	Lexical Node Table (31 nodes)
k	Safe Nodes!
l	Predefined Symbols (109 names, 574 characters)
m	Field Symbols
n	New Symbol Allocation Classes

SHIFT BACK to return tables to -trw6-

```
Class Members....
 none -
    1 - X
    2 - (
    3 - .
    4 - 1 2 3 4 5 6 7 8
    5 - )
    6 - -
    7 - $
    8 - *
    9 - Z
   1Ø - , B
   11 - S
   12 - 9
   13 - V
   14 - .
   15 - Ø
   16 - + ; A P
   17 - ¢ < | & ! ¬ / % _ > ? : # @ ' = " C D E F G H I J K L M
        N O Q R T U W Y ?
   18 -
   19 - C1 C2 C3 C4 C5 C6
   2Ø - C7 C8 C9 C1Ø C11 C12 C13 C14 C15 C16 C17 C18 C19 C2Ø C21
        C22 C23 C24 C25 C26 C27 C28 C29 C3Ø C31 C32 C33 C34 C35
        C36 C37 C38 C39 C4Ø C41 C42 C43 C44 C45 C46 C47 C48 C49 C
        5Ø C51 C52 C53 C54 C55 C56 C57 C58 C59 C6Ø C61 C62 C63 C6
        4
```

B.2.2 Lexical node table

NODE	CHARACTER CLASS →									
↓↓	1	(X)	2	(()	3	(.)	4	(1)	5	())
Ø	32	E	2	P	11	X	32	E	32	E
1	1		2	P	1	X	19	E	19	E
2	4	E	4	E	4	E	3		4	E
3	12	E	12	E	12	E	3		18	R
4	2Ø	E	2	P	2Ø	E	2Ø	E	2Ø	E
5	21	E	2	P	21	E	21	E	21	E
6	22	E	2	P	22	E	22	E	22	E
7	23	E	2	P	6	X	23	E	23	E
8	24	E	2	P	6	X	24	E	24	E
9	25	E	2	P	2	X	25	E	25	E
1Ø	26	E	2	P	6	X	26	E	26	E
11	27	E	2	P	2	X	27	E	27	E
12	34	E	34	E	6	X	34	E	34	E
13	35	E	35	E	6	X	35	E	35	E
14	36	E	36	E	2	X	36	E	36	E
15	37	E	37	E	6	X	37	E	37	E
16	33	E	33	E	2	X	33	E	33	E
17	39	E	39	E	39	E	39	E	39	E
18	28	E	2	P	14	X	28	E	28	E
19	29	E	2	P	14	X	29	E	29	E
2Ø	3Ø	E	2	P	1Ø	X	3Ø	E	3Ø	E
21	31	E	2	P	14	X	31	E	31	E
22	32	E	2	P	1Ø	X	32	E	32	E
23	27	E	2	P	3	X	27	E	27	E
24	33	E	33	E	3	X	33	E	33	E
25	39	E	39	E	39	E	39	E	39	E
26	1		51	E	26		51	E	51	E
27	4Ø	E	4Ø	E	4Ø	E	4Ø	E	4Ø	E
28	41	E	2	P	3	X	41	E	41	E
29	42	E	42	E	3	X	42	E	42	E
3Ø	43	E	2	P	11	X	43	E	43	E
31	38	E	38	E	3	X	38	E	38	E

NODE ↓	CHARACTER CLASS → 6 (-)		7 ($)		8 (*)		9 (Z)		10 (,)	
0	32	E	32	E	32	E	32	E	22	
1	19	E	19	E	19	E	19	E	54	E
2	4	E	4	E	4	E	4	E	4	E
3	12	E	12	E	12	E	12	E	12	E
4	7		5		9		10		6	
5	21	E	8		9		10		6	
6	22	E	22	E	9		10		6	
7	7		23	E	23	E	23	E	7	
8	24	E	8		24	E	24	E	8	
9	25	E	25	E	9		25	E	9	
10	26	E	26	E	26	E	10		10	
11	27	E	27	E	27	E	27	E	11	
12	18		34	E	34	E	34	E	12	
13	35	E	19		35	E	35	E	13	
14	36	E	36	E	20		36	E	14	
15	37	E	37	E	37	E	21		15	
16	33	E	33	E	33	E	33	E	16	
17	39	E	39	E	20		21		17	
18	18		28	E	28	E	28	E	18	
19	29	E	19		29	E	29	E	19	
20	30	E	30	E	20		30	E	20	
21	31	E	31	E	31	E	21		21	
22	32	E	32	E	32	E	32	E	22	
23	27	E	27	E	27	E	27	E	11	
24	33	E	33	E	33	E	33	E	16	
25	39	E	39	E	20		21		17	
26	4		5		9		10		6	
27	40	E	40	E	40	E	40	E	40	E
28	41	E	41	E	41	E	41	E	41	E
29	42	E	42	E	42	E	42	E	42	E
30	43	E	43	E	43	E	43	E	43	E
31	38	E	38	E	38	E	38	E	16	

NODE ↓↓	CHARACTER CLASSES→ 11 (S)	12 (9)	13 (V)	14 (.)	15 (Ø)
Ø	32 E	Ø	32 E	11 X	22
1	19 E	54 E	19 E	1 X	54 E
2	4 E	3	4 E	4 E	2
3	12 E	3	12 E	12 E	3
4	2Ø E	11	17	17	6
5	21 E	11	17	17	6
6	22 E	11	17	17	6
7	23 E	11	12	12	7
8	24 E	11	13	13	8
9	25 E	11	14	14	9
1Ø	26 E	11	15	15	1Ø
11	27 E	11	16	16	11
12	34 E	22	34 E	6 X	12
13	35 E	22	35 E	6 X	13
14	36 E	22	36 E	2 X	14
15	37 E	22	37 E	6 X	15
16	33 E	22	33 E	2 X	16
17	39 E	22	39 E	39 E	17
18	28 E	28 E	28 E	14 X	18
19	29 E	29 E	29 E	14 X	19
2Ø	3Ø E	3Ø E	3Ø E	1Ø X	2Ø
21	31 E	31 E	31 E	14 X	21
22	32 E	22	32 E	1Ø X	22
23	27 E	23	24	31	11
24	33 E	Ø	33 E	3 X	22
25	39 E	Ø	39 E	39 E	17
26	27	23	25	17	6
27	4Ø E	28	29	4Ø E	4Ø E
28	41 E	28	29	3 X	41 E
29	42 E	3Ø	42 E	3 X	42 E
3Ø	43 E	3Ø	43 E	11 X	43 E
31	38 E	22	38 E	38 E	16

NOTE: CHARACTER CODES→

↓↓	16 (+)	17 (¢)	18 (	19 ()	20 (
Ø	32 E	53 E	Ø	52 E	Ø
1	54 E	53 E	1	52 E	1
2	4 E	53 E	59 E	52 E	2
3	12 E	53 E	59 E	52 E	3
4	2Ø E	53 E	7	52 E	4
5	21 E	53 E	8	52 E	5
6	22 E	53 E	6	52 E	6
7	23 E	53 E	7	52 E	7
8	24 E	53 E	8	52 E	8
9	25 E	53 E	9	52 E	9
1Ø	26 E	53 E	1Ø	52 E	1Ø
11	27 E	53 E	11	52 E	11
12	34 E	53 E	59 E	52 E	12
13	35 E	53 E	59 E	52 E	13
14	36 E	53 E	59 E	52 E	14
15	37 E	53 E	59 E	52 E	15
16	33 E	53 E	59 E	52 E	16
17	39 E	53 E	59 E	52 E	17
18	28 E	53 E	18	52 E	18
19	29 E	53 E	19	52 E	19
2Ø	3Ø E	53 E	2Ø	52 E	2Ø
21	31 E	53 E	21	52 E	21
22	32 E	53 E	22	52 E	22
23	27 E	53 E	23	52 E	23
24	33 E	53 E	59 E	52 E	24
25	39 E	53 E	59 E	52 E	25
26	56 E	53 E	59 E	26	26
27	4Ø E	53 E	59 E	52 E	27
28	41 E	53 E	28	52 E	28
29	42 E	53 E	59 E	52 E	29
3Ø	43 E	53 E	3Ø	52 E	3Ø
31	38 E	53 E	59 E	52 E	31

The following nodes are currently designated as "safe" nodes:

26

Press [EDIT] to add nodes to this list.
[ERASE] to delete nodes from this list
[BACK] to return to index page
[HELP] to for an explanation of "safe" nodes

When the user types a CR the editor must know whether to pad the line with blanks or cut it short. To determine this the editor feeds blanks to Lexi until it enters a node which been designated as a safe place to terminate the line. Typical "safe" nodes would be the initial node and the node used to collect a string of blanks.

APPENDIX C

OTHER DOCUMENTATION

Supplementary documentation for the COBOL system, produced by auxiliary programs, is included in this appendix. Only a sample of the transition diagrams is included, since they include the same information as presented in the lexical node tables.

C.1 Transition diagram sample

Transition Diagram Generator

To display lexical transition diagrams for use with tables of the Computer Assisted Programming System.

Explanation of Symbols

lexical node number

procedure call

error message number

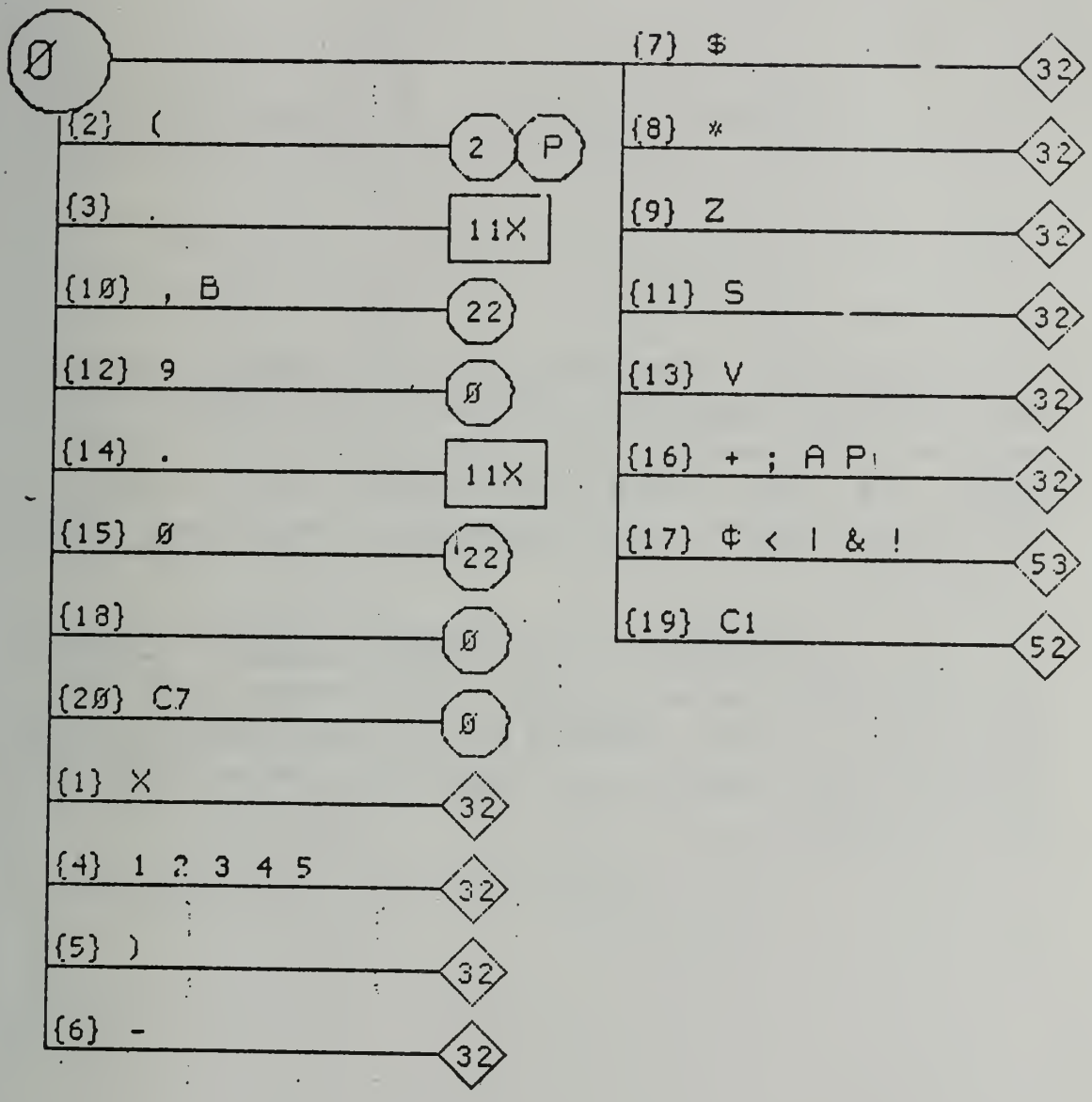
syntactic class of an accepted token

lexical character class returned by subroutine

{ } character class number, then first 5 class members

N	Non-extendible token
X	eXtendible token
I	Ignore token (do not send to parser)
Fn	Generate field #n (1, 2, or 3)
FP	Field starts with next column
FB	Field begins with current column
FE	Field ends with current column
FD	Field ends with previous column
Cn	Column #n

Node Ø: {9} V {9}+

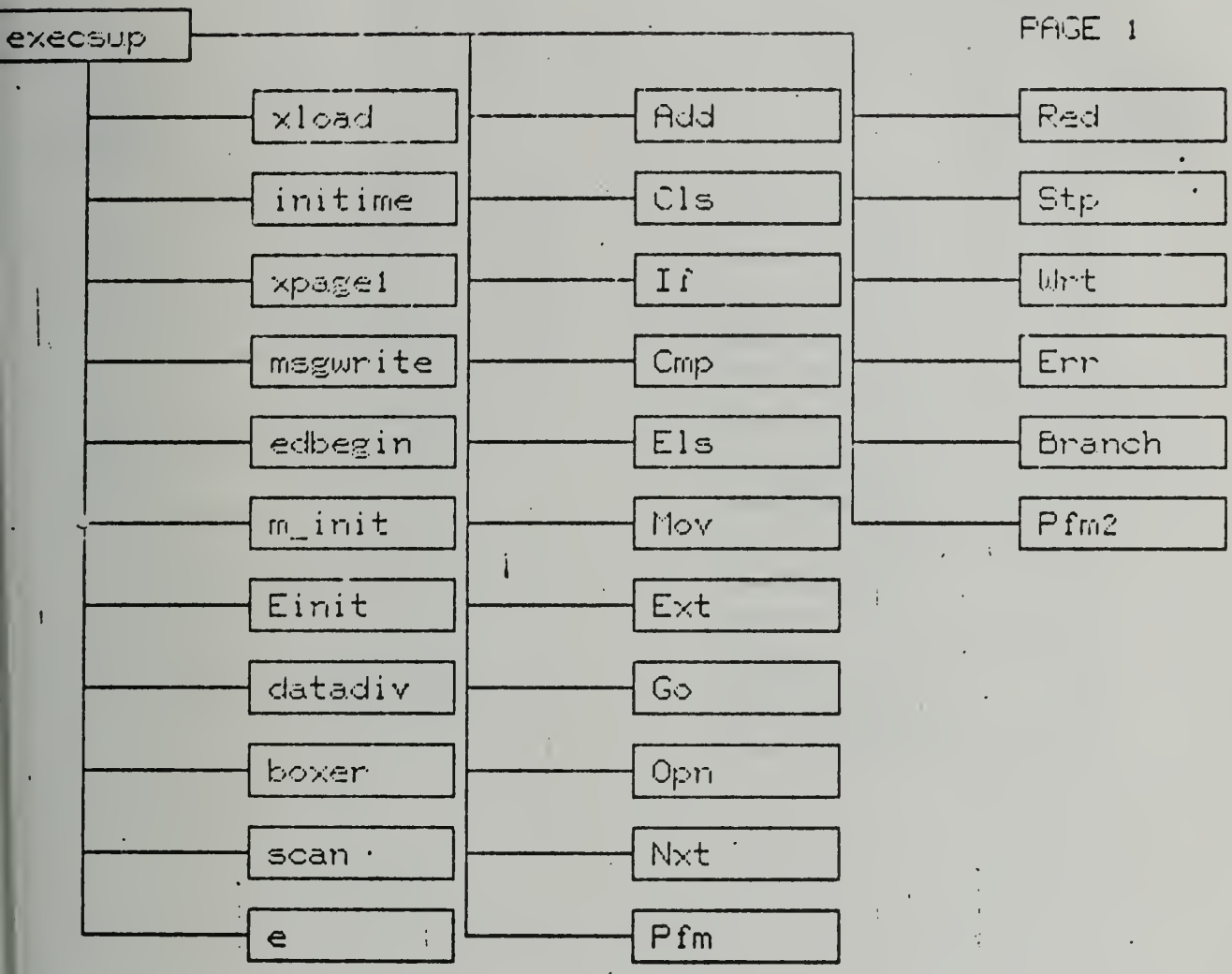

Ø
{2} (
2
P
{3} .
11X
{10} , B
22
{12} 9
Ø
{14} .
11X
{15} Ø
22
{18}
Ø
{20} C7
Ø
{1} X
32
{4} 1 2 3 4 5
32
{5})
32
{6} -
32
{7} $
32
{8} %
32
{9} Z
32
{11} S
32
{13} V
32
{16} + ; A P
32
{17} ¢ < | & !
53
{19} C1
52

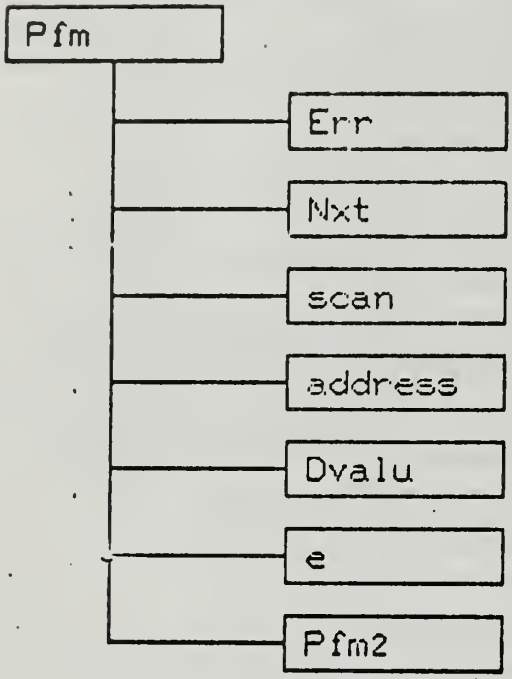
C.2 Flowchart of "cobolrun"

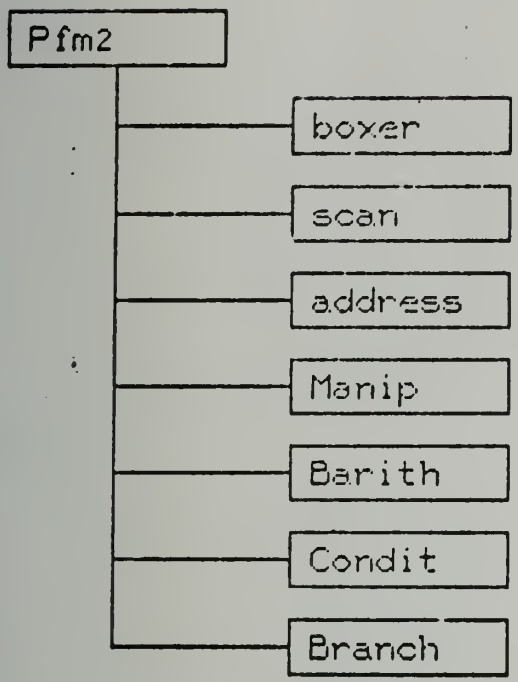
"cobolrun" flowchart

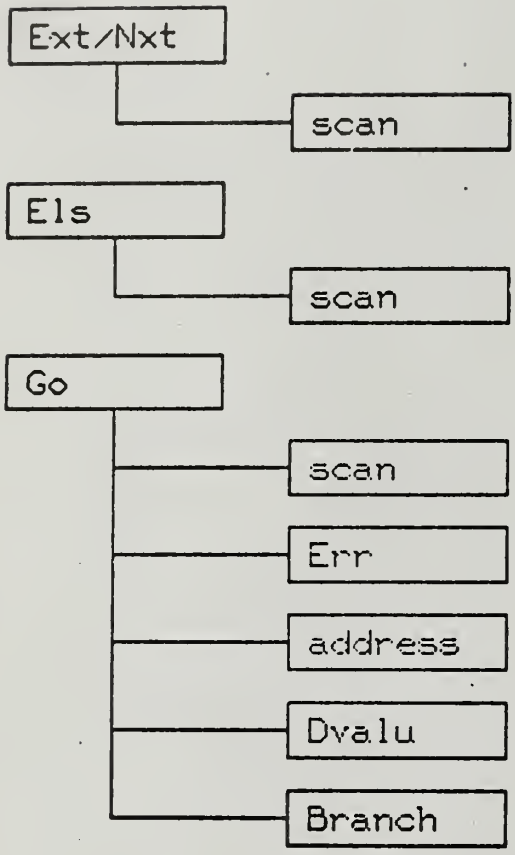
Except for the units of "trw" used by cobolrun, each unit of "cobolrun" appears on at least one page of this lesson. If it appears on more than 1 page, its "home" page (the one you get by typing TERM unit-name) is the highest numbered-page that it appears on.

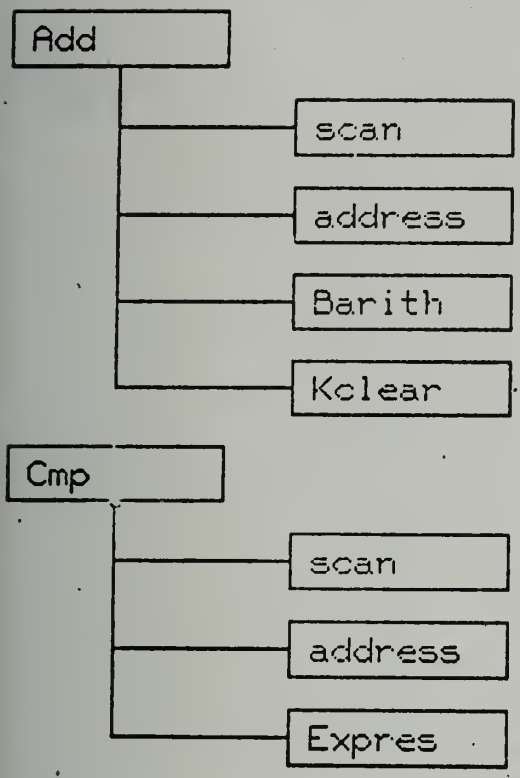
The flowcharts are arranged by pages according to their calling sequences. This means that the called unit never appears on a lower-numbered page than the calling unit (a "call" may be a TUTOR do, join or goto).


execsup
xload
initime
xpage1
msgwrite
edbegin
m_init
Einit
datadiv
boxer
scan
e
Add
Cls
If
Cmp
Els
Mov
Ext
Go
Opn
Nxt
Pfm
Red
Stp
Wrt
Err
Branch
Pfm2
PAGE 1

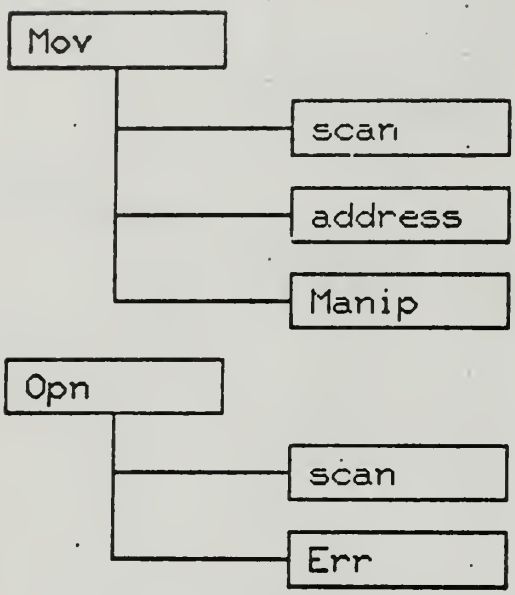

Pfm
Err
Nxt
scan
address
Dvalu
e
Pfm2

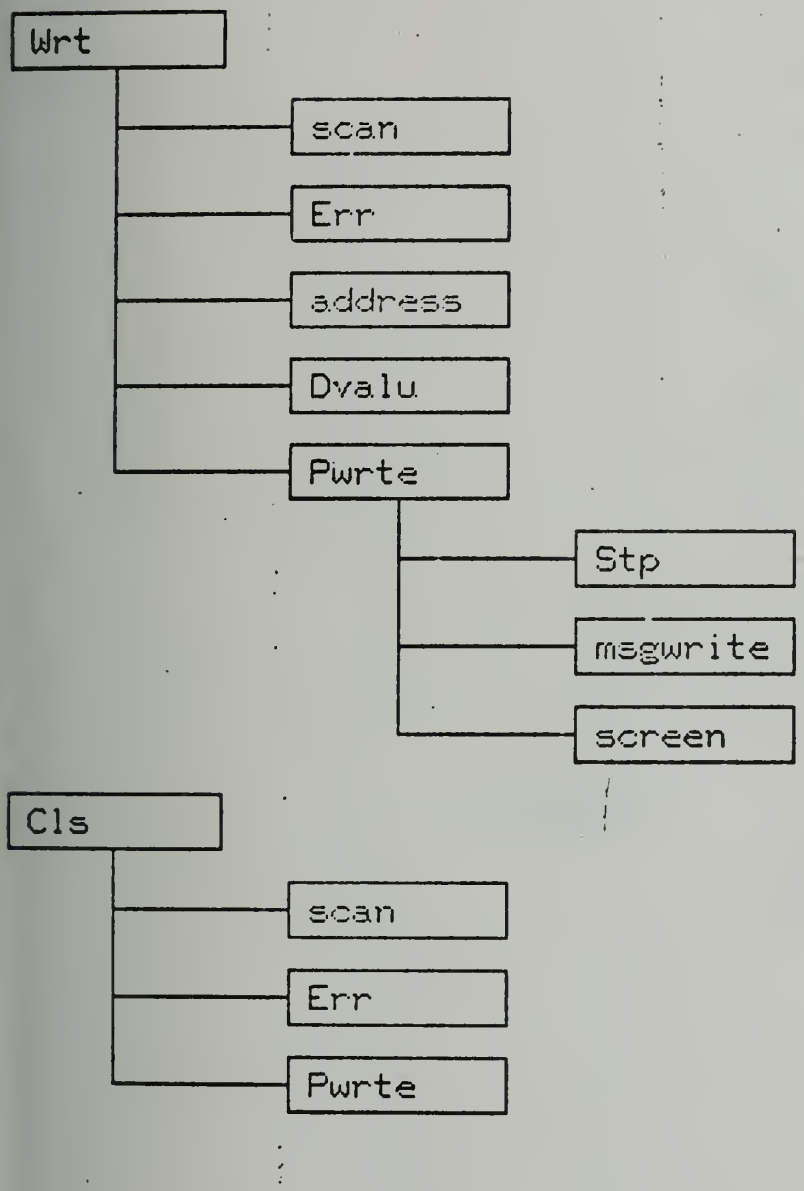

Pfm2
boxer
scan
address
Manip
Barith
Condit
Branch

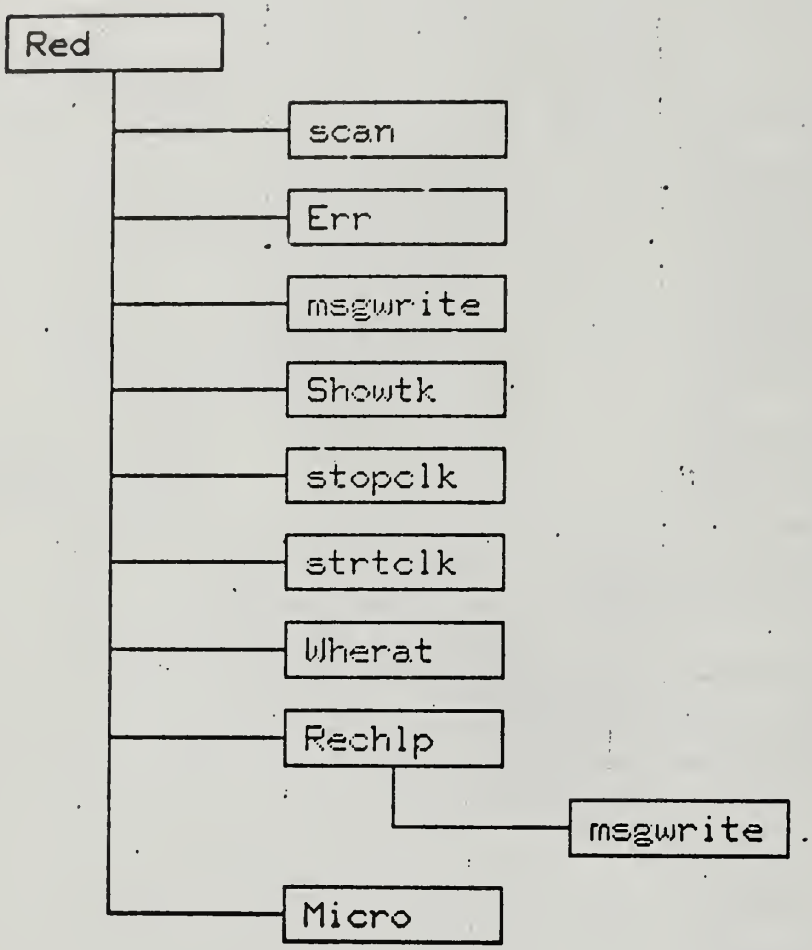

Ext/Nxt
scan
Els
scan
Go
scan
Err
address
Dvalu
Branch

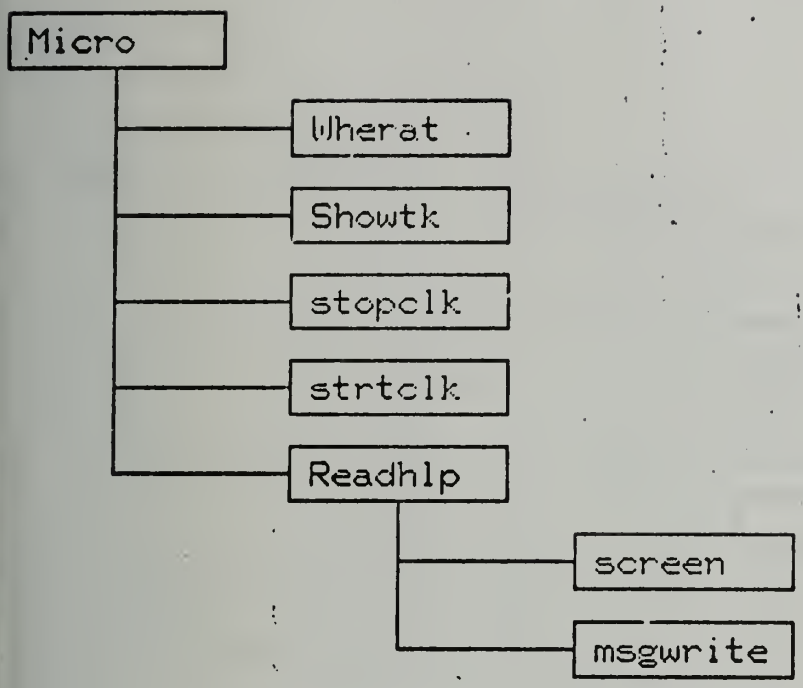


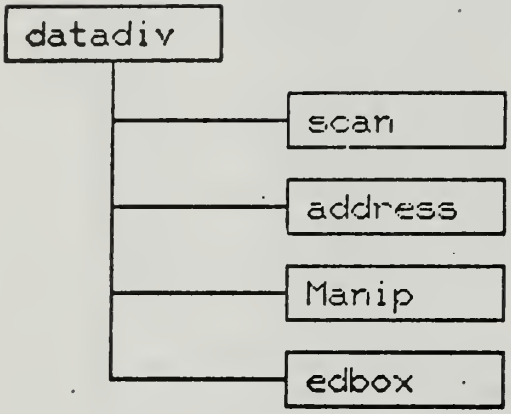
PAGE 6


Mov
scan
address
Manip
Opn
scan
Err

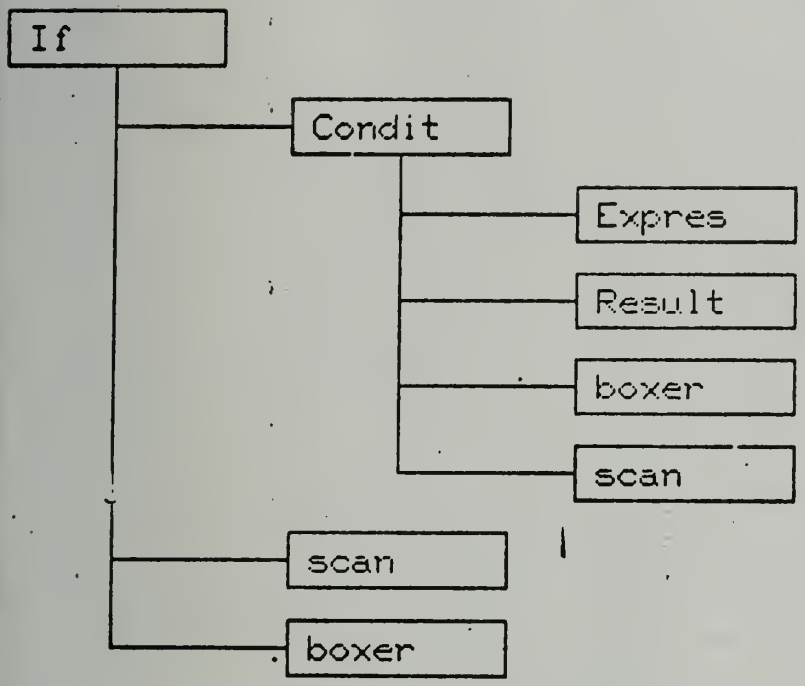

Wrt
scan
Err
address
Dvalu
Pwrte
Stp
msgwrite
screen
Cls
scan
Err
Pwrte

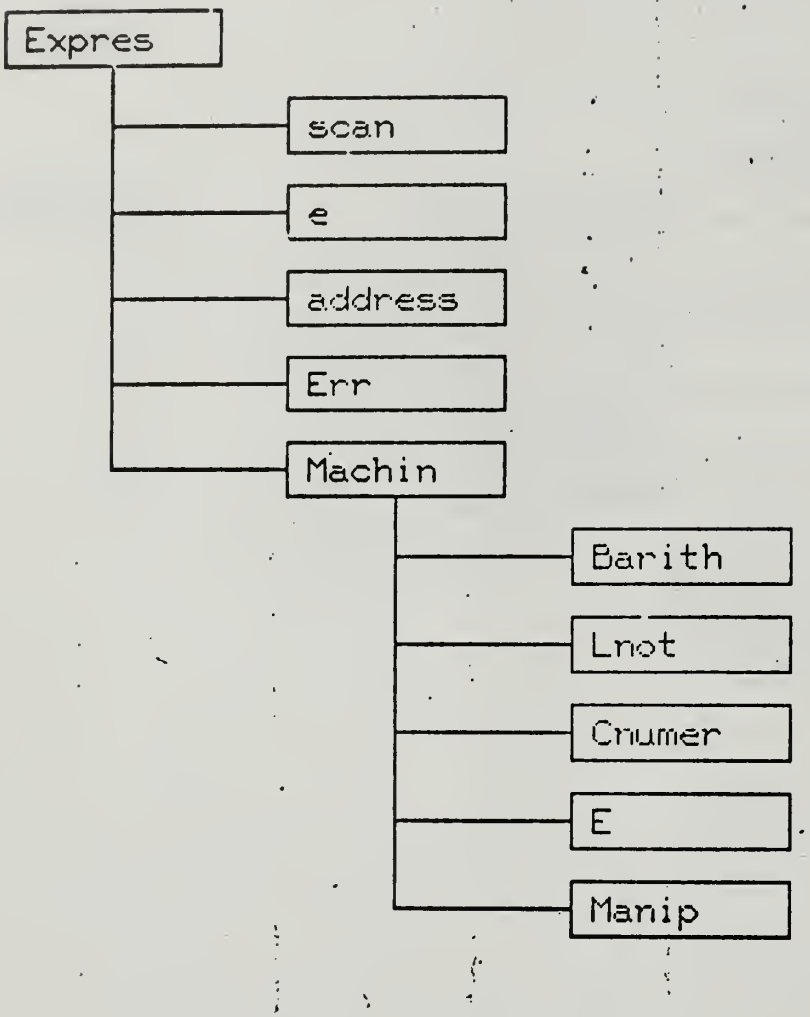

Red
scan
Err
msgwrite
Showtk
stopclk
strtclk
Wherat
Rechlp
msgwrite
Micro

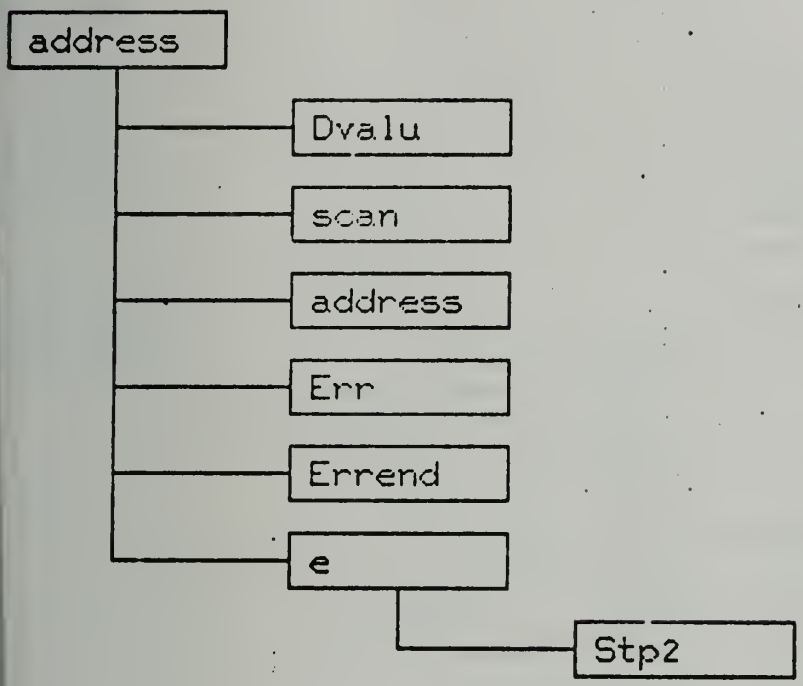

Micro
Wherat
Showtk
stopclk
strtclk
Readhlp
screen
msgwrite



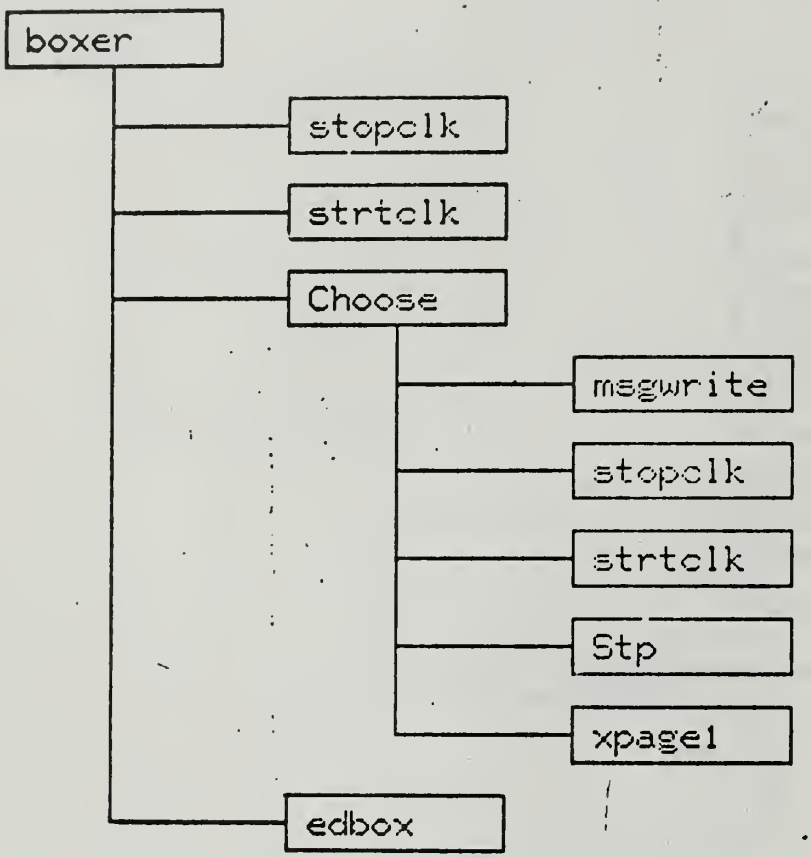
PAGE 11

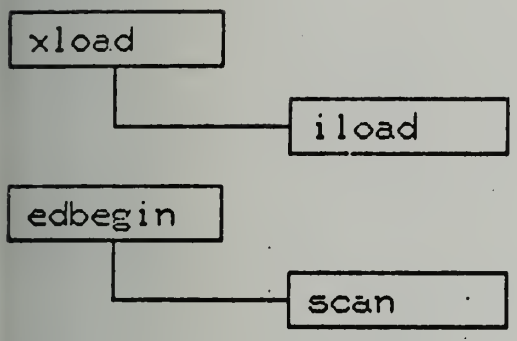

If
Condit
Expres
Result
boxer
scan
scan
boxer


Expres
scan
e
address
Err
Machin
Barith
Lnot
Cnumer
E
Manip

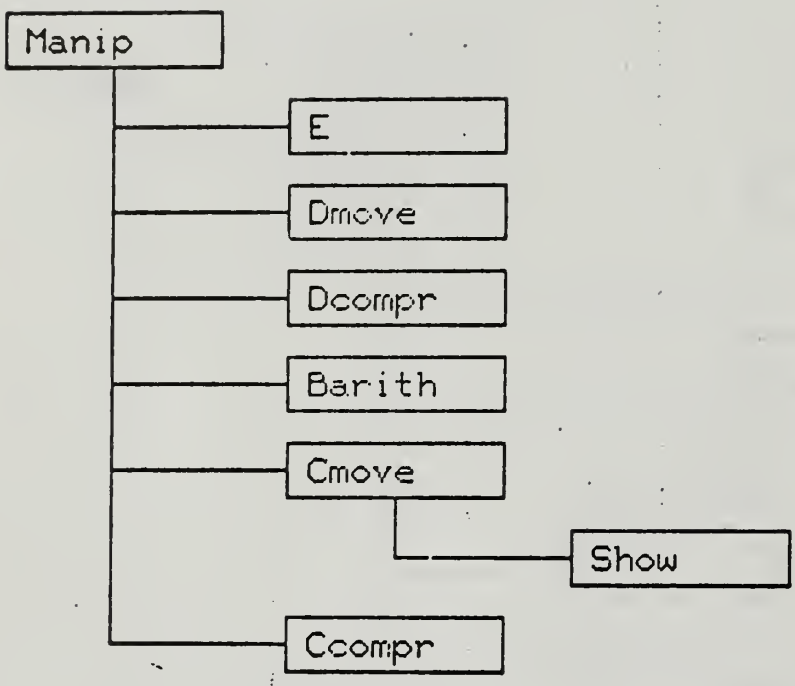


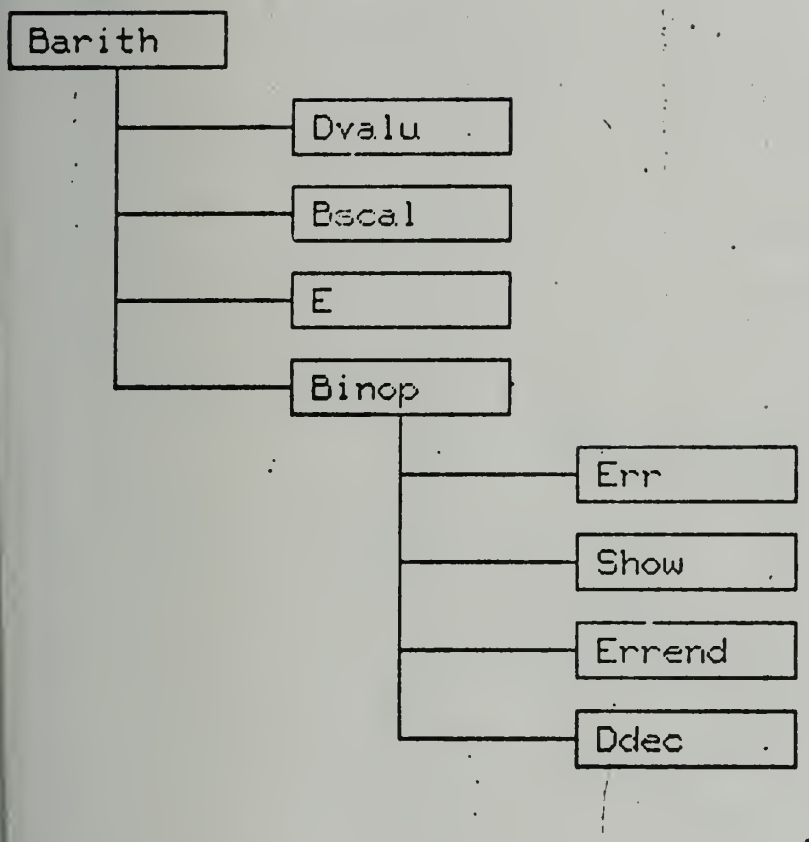
PAGE 15


boxer
stopclk
strtclk
Choose
msgwrite
stopclk
strtclk
Stp
xpage1
edbox

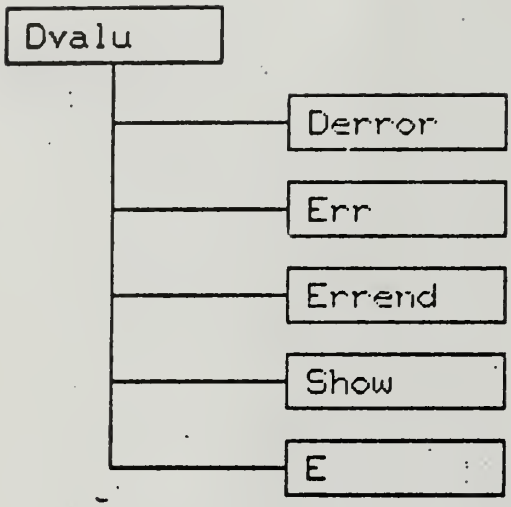


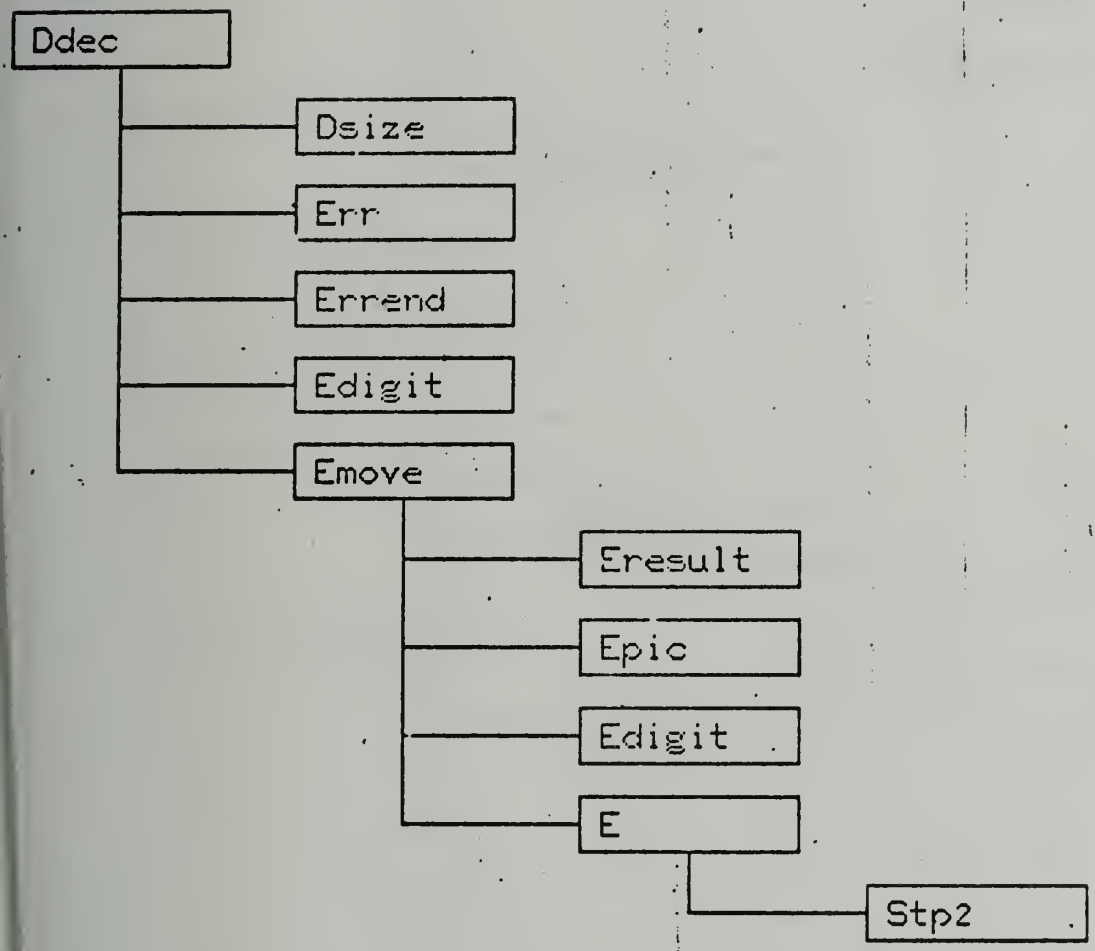
PAGE 17

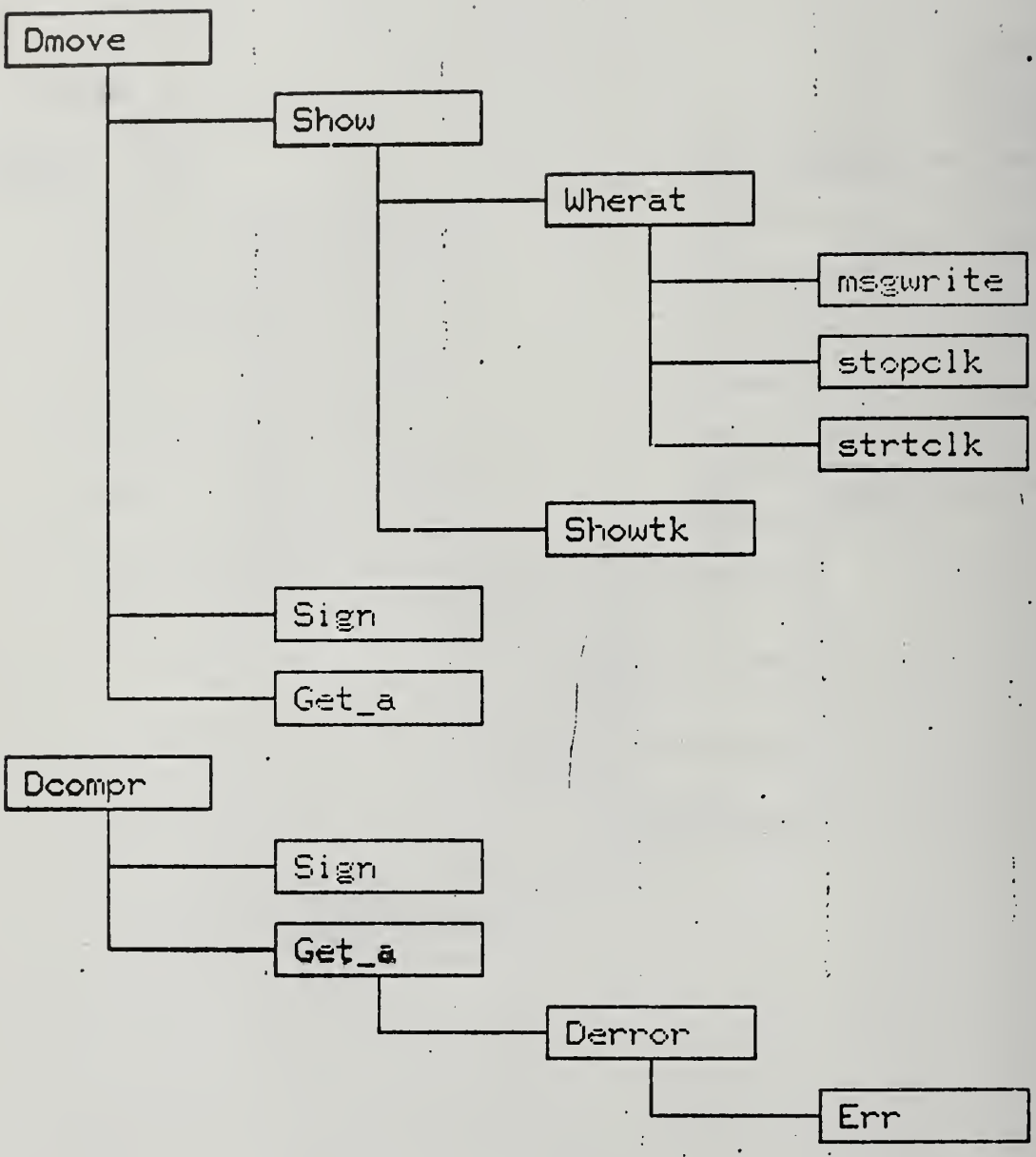

Manip
E
Dmove
Dcompr
Barith
Cmove
Show
Ccompr



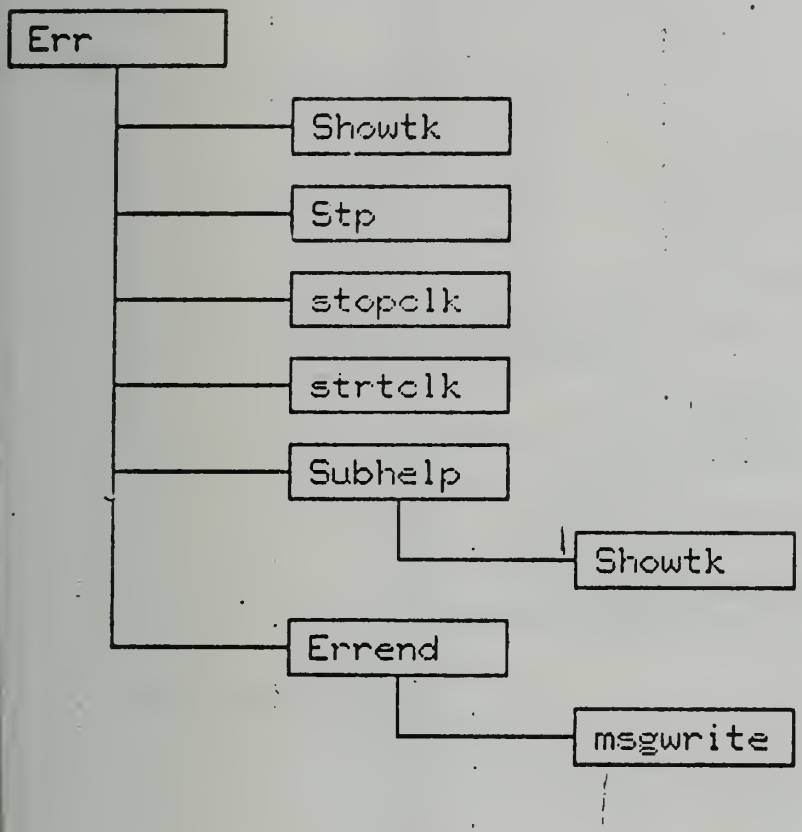
PAGE 20

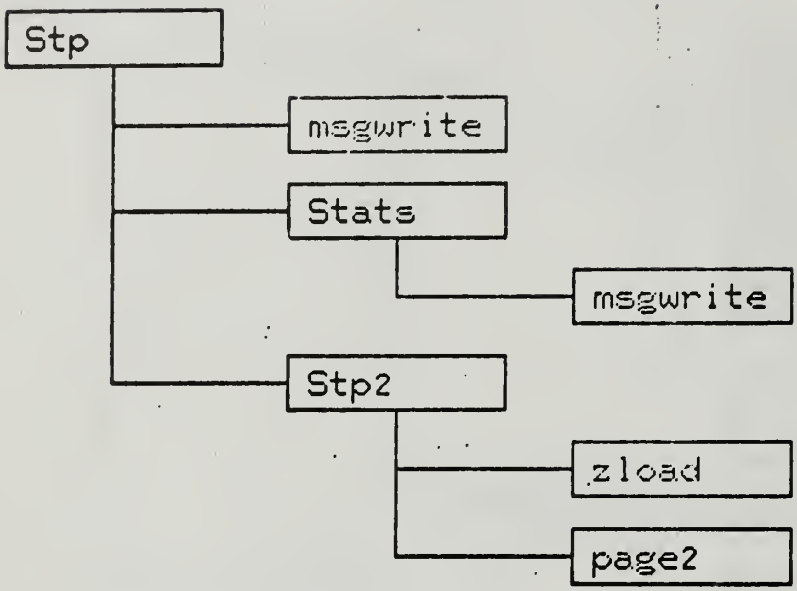

Dvalu
Derror
Err
Errend
Show
E


Ddec
Dsize
Err
Errend
Edigit
Emove
Eresult
Epic
Edigit
E
Stp2

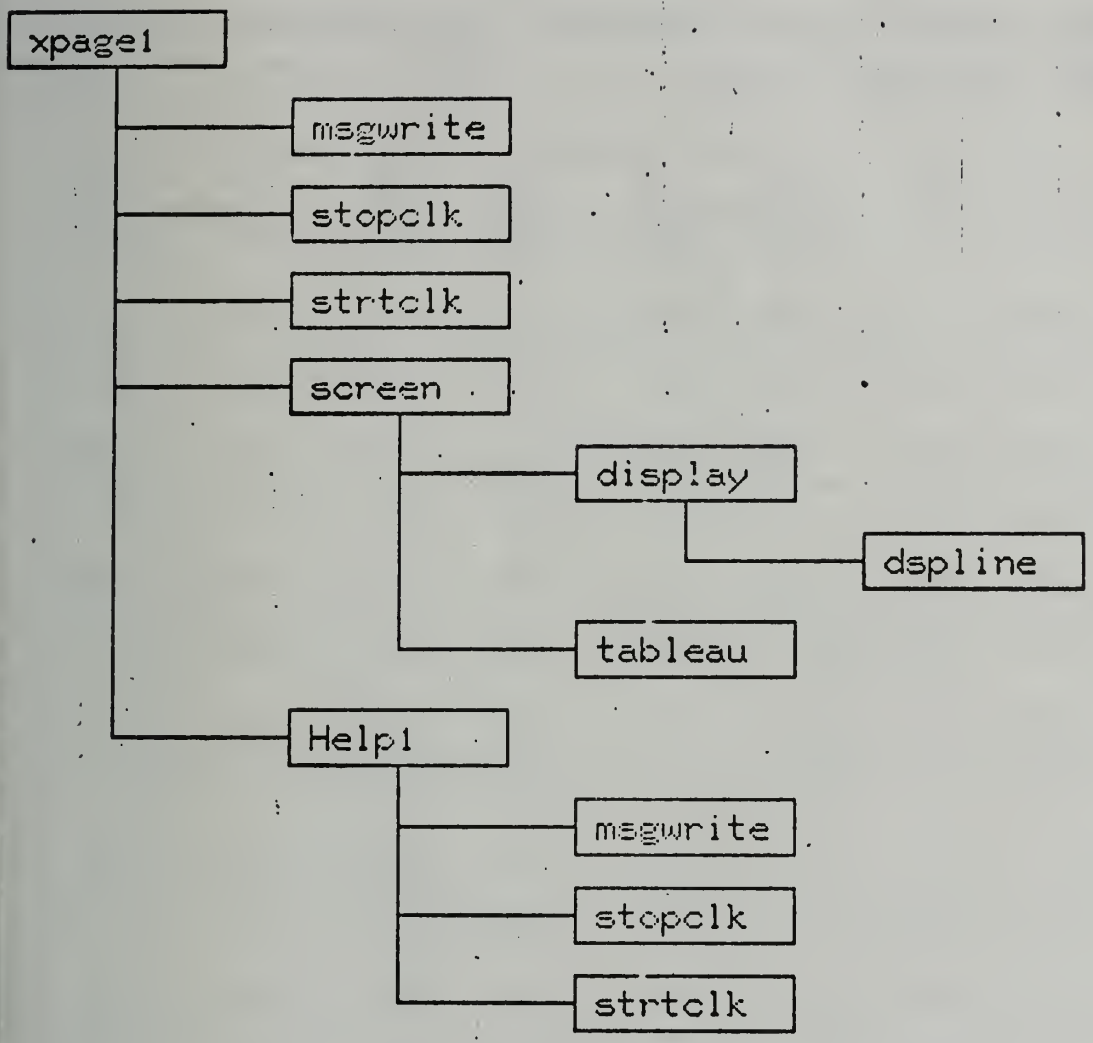

Dmove
Show
Wherat
msgwrite
stopclk
strtclk
Showtk
Sign
Get_a
Dcompr
Sign
Get_a
Derror
Err

PAGE 24


Err
Showtk
Stp
stopclk
strtclk
Subhelp
Showtk
Errend
msgwrite



PAGE 25


xpage1
msgwrite
stopclk
strtclk
screen
display
dspline
tableau
Help1
msgwrite
stopclk
strtclk

C.3 Mode table diagram

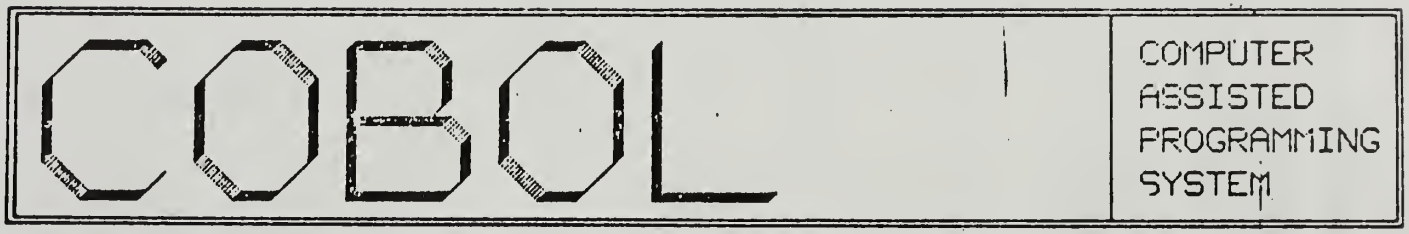
NO: illegal move or compare; DEC: decimal mode | move
CHR: character mode; BIN: binary mode | cmpr

FROM \ TO	GR	AN	NE	XL	BI	ED	ZR	NL
GR	CHR CHR	CHR CHR	CHR CHR	NO CHR	CHR NO	CHR CHR	NO CHR	NO CHR
AN	CHR CHR	CHR CHR	BIN CHR	NO CHR	NO NO	DEC CHR	NO CHR	NO CHR
NE	CHR CHR	CHR CHR	NO CHR	NO CHR	NO NO	NO CHR	NO CHR	NO CHR
XL	CHR CHR	CHR CHR	NO CHR	NO NO	NO NO	NO CHR	NO NO	NO NO
BI	CHR NO	NO NO	BIN NO	NO NO	BIN BIN	BIN BIN	NO BIN	NO BIN
ED	CHR CHR	CHR CHR	BIN CHR	NO CHR	BIN BIN	DEC DEC	NO DEC	NO DEC
ZR	CHR CHR	CHR CHR	BIN CHR	NO NO	BIN BIN	DEC DEC	NO NO	NO NO
NL	NO CHR	NO CHR	BIN CHR	NO NO	BIN BIN	DEC DEC	NO NO	NO NO

APPENDIX D

SAMPLE COBOL PROGRAM

In this appendix several displays that might be seen by the student as he uses the COBOL system are shown. This is far from a complete set; the student may also see error messages, "help" information in the interpreter and the information contained in appendix A.



Press HELP for instructions on how to use this lesson (you'll need this if it is your first time in this lesson);

Or press NEXT to begin.

YOUR WORKSPACE IS EMPTY

Press...	To...
NEXT	Write a new COBOL program.
SHIFT NEXT	Write in another language.
EDIT	Edit one of your old programs.
ERASE	Erase one of your programs.
LAB	Execute one of your programs.
DATA	See a list of programs on file.
SHIFT HELP	Return to the lesson guide.

```
.FD..ACCOUNT-FILE.LABEL.RECORDS.OMITTED..
.Ø1..ACCOUNT..
.....Ø3..A-NUMBER.....PIC.9(5)..
.....Ø3..A-OLD-BAL.....PIC.9(6)V99..
.....Ø3..A-SER-CHG.....PIC.9(6)V99..
.FD..BALANCE-FILE.LABEL.RECORDS.OMITTED..
.Ø1..B-REC..Ø3.B-OLD-BAL.PIC.$(8),$$$.$$..
.....Ø3..B-NEW-BAL.....PIC.$(8),$$$.$$..
.....Ø3..B-DIFF.....PIC.-(8),---.--..
.WORKING-STORAGE.SECTION..
.Ø1..TOTALS.COMP..
....Ø3.W-OLD-BAL.....PIC.S9(6)V99.VALUE.ZERO..
.....Ø3.W-NEW-BAL.....PIC.S9(6)V99.VALUE.ZERO..
.PROCEDURE.DIVISION..
.OPENER..OPEN.INPUT.ACCOUNT-FILE.OUTPUT.BALANCE-FILE..
.READER..READ.ACCOUNT-FILE.END
.........MOVE.W-OLD-BAL.TO.B-OLD-BAL
.........MOVE.W-NEW-BAL.TO.B-NEW-BAL
.........SUBTRACT.W-OLD-BAL.FROM.W-NEW-BAL.GIVING.B-DIFF
.........WRITE.B-REC..STOP.RUN..
.....ADD.A-OLD-BAL.TO.W-OLD-BAL
.....COMPUTE.W-NEW-BAL.=.W-NEW-BAL.+.A-OLD-BAL.-.A-SER-CHG
.....GO.TO.READER..
.END._
```

WORKSPACE CONTAINS TEXT

(Press DATA for details)

Press...	To...
NEXT	Edit your workspace some more.
LABEL	Execute your workspace as a program.
SHIFT BACK	Clear your workspace.
COPY	Copy your workspace into the file.
SHIFT COPY	Replace 'cobolk' with workspace.
ERASE	Erase file 'cobolk' from the file.

```
.FD..ACCOUNT-FILE.LABEL.RECORDS.OMITTED..
.Ø1..ACCOUNT..
.....Ø3..A-NUMBER.....PIC.9(5)..
.....Ø3..A-OLD-BAL.....PIC.9(6)V99..
.....Ø3..A-SER-CHG.....PIC.9(6)V99..
.FD..BALANCE-FILE.LABEL.RECORDS.OMITTED..
.Ø1..B-REC..Ø3.B-OLD-BAL.PIC.$(6),$$$.$$..
.....Ø3..B-NEW-BAL.....PIC.$(8),$$$.$$..
.....Ø3..B-DIFF.....PIC.-(8),---.--..
.WORKING-STORAGE.SECTION..
.Ø1..TOTALS.COMP..
.....Ø3.W-OLD-BAL.....PIC.S9(6)V99.VALUE.ZERO..
.....Ø3.W-NEW-BAL.....PIC.S9(6)V99.VALUE.ZERO..
.PROCEDURE.DIVISION..
.OPENER..OPEN.INPUT.ACCOUNT-FILE.OUTPUT.BALANCE-FILE..
.READER..READ.ACCOUNT-FILE.END
.........MOVE.W-OLD-BAL.TO.B-OLD-BAL
.........MOVE.W-NEW-BAL.TO.B-NEW-BAL
.........SUBTRACT.W-OLD-BAL.FROM.W-NEW-BAL.GIVING.B-DIFF
.........WRITE.B-REC..STOP.RUN..
.....ADD.A-OLD-BAL.TO.W-OLD-BAL
.....COMPUTE.W-NEW-BAL.=.W-NEW-BAL.+.A-OLD-BAL.-.A-SER-CHG
.....GO.TO.READER..
.END.
```

```
W-OLD-BAL ⇐ 2495Ø19
W-NEW-BAL ⇐ 2494294
```

COBOL EXECUTION STATISTICS

Variable Storage Used:	76 characters
% of Storage Used:	4 %
Statements Executed:	22
Lines Outputted:	1
Total Execution Time:	Ø.947 seconds
Average Time/statement:	Ø.ØØ43Ø seconds
Average Statements/second:	23.2

APPENDIX E

PROGRAM LISTINGS

In order to reduce the size of the report version of this thesis, the program listings included in Appendix E of the original thesis have been excluded.

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-75-685	2.	3. Recipient's Accession No.
Title and Subtitle	AN INTERACTIVE COBOL SYSTEM FOR PLATO		5. Report Date January, 1975
			6.
Author(s)	Robert D. Barnett		8. Performing Organization Rept. No.
Performing Organization Name and Address	University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801		10. Project/Task/Work Unit No. 11. Contract/Grant No.
Sponsoring Organization Name and Address	University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801		13. Type of Report & Period Covered M.S. Thesis - 1975 14.

Supplementary Notes

Abstracts

An interactive COBOL compiler system for the PLATO computer-based education system is described. The system has two major components: an editor and an interpreter. The editor uses language-independent routines developed for PLATO, driven by tables. The interpreter acts on a representation of the COBOL program to execute the procedural statements. An explanation of the implemented COBOL subset is included.

ey Words and Document Analysis. 17a. Descriptors

Computing Reviews categories 3.51, 4.13 and 4.22.

COBOL
Compilers
Diagnostic routines
Interpreter routines
Man machine systems

Identifiers/Open-Ended Terms

Computer assisted instruction
Data processing training

OSATI Field/Group

Availability Statement

Release unlimited

19. Security Class (This Report) UNCLASSIFIED
21. No. of Pages 140
20. Security Class (This Page UNCLASSIFIED
22. Price

M TIS-35 (10-70)

USCOMM-DC 40329-P71

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.685-690(1974
Program manual : NOR network pruning pro
3 0112 088401598